CodeWarrior®
C, C++, and Assembly Language Reference

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 2201 Donley Drive, Suite 310 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Mail order	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
America Online	keyword: `Metrowerks`
CompuServe	goto `Metrowerks`

Table of Contents

1

Introduction

This manual describes how the Metrowerks C and C++ compilers implement the C and C++ standards and its in-line assembler.

Overview of the C/C++/ASM Reference

This manual describes how the Metrowerks C and C++ compilers implement the C and C++ standards and its in-line assembler. Each chapter begins with an overview.

Table 1.1 What's in this manual

This chapter…	Documents…
Overview of C and C++ Language Notes	How Metrowerks C implements the C standard. It also describes the parts of C++ that it shares with C.
Overview of C++ Language Notes	How Metrowerks C++ implements the parts of the C++ standard that are unique to C++. It also describes how to use templates and exception handling.
Overview of 68K Assembler Notes	How to use the 68K inline assembler, which is part of Metrowerks C and C++, to include assembly code in your program.
Overview of PowerPC Assembler Notes	How to use the PowerPC inline assembler, which is part of Metrowerks C and C++, to include assembly code in your program.
Overview of MIPS Assembler Notes	How to use the MIPS inline assembler, which is part of Metrowerks C and C++, to include assembly code in your program.

This chapter…	Documents…
Overview of Win32/x86 Assembler Notes	How to use the Win32/x86 inline assembler, which is part of Metrowerks C and C++, to include assembly code in your program.
Overview of Pragmas and Predefined Symbols	The pragma statement, which lets you change your program's options from your source code. It also describes the preprocessor function `__option()`, which lets you test the setting of many pragmas and options, and the predefined symbols that Metrowerks C and C++ use.

Conventions Used in This Manual

This manual includes syntax examples that describe how to use certain statements, such as the following:

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

Table 1.2 describes how to interpret these statements.

Table 1.2 Understanding Syntax Examples

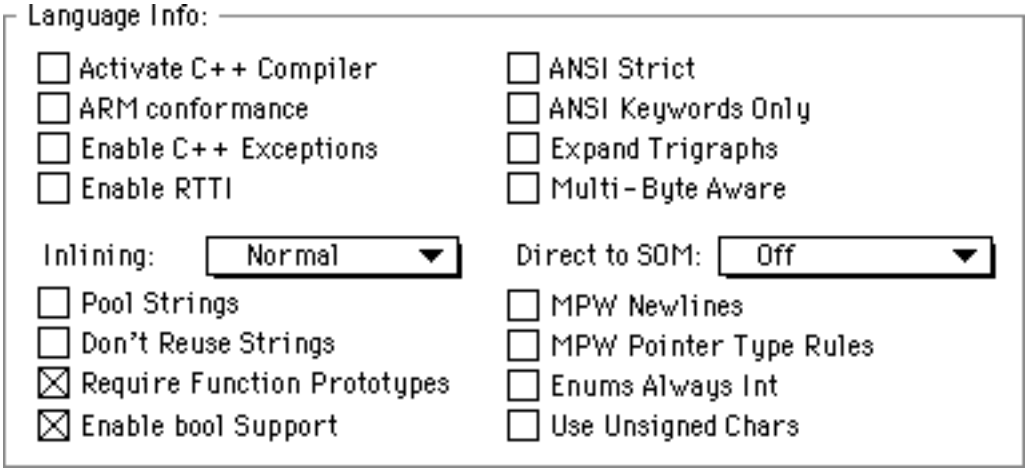
If the text looks like…	Then…
`literal`	Include it in your statement exactly as it's printed.
metasymbol	Replace the symbol with an appropriate value. The text after the syntax example describes what the appropriate values are.
`a | b | c`	Use one and only one of the symbols in the statement: either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

The C/C++ Project Settings Panels

This section describes where to find information on the C/C++ Language and C/C++ Warnings settings panels.

This is the C/C++ Language settings panel:

Figure 1.1 The C/C++ Languages Settings Panel



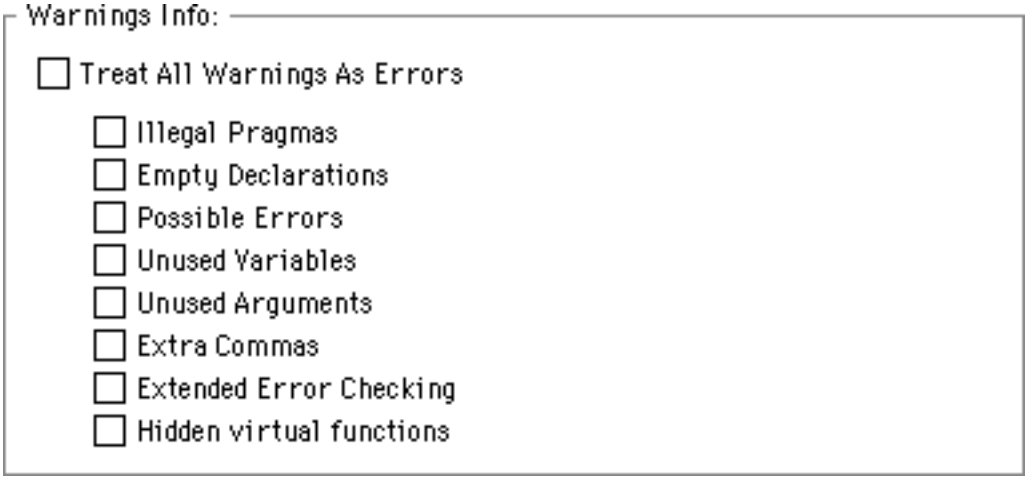
This table describes where to find more information on its options:

This option…	Is described here…
Activate C++ Compiler	"Using the C++ compiler always" on page 93
ARM Conformance	"Enforcing strict ARM conformance" on page 94
Enable C++ Exceptions	"Allowing exception handling" on page 96
Enable RTTI	"Using Run-Time Type Information (RTTI)" on page 96
Inlining	"Inlining functions" on page 52
Pool Strings	"Pooling strings" on page 56
Don't Reuse Strings	"Reusing strings" on page 58
Require Function Prototypes	"Requiring prototypes" on page 54
Enable bool Support	"Using the bool type" on page 96
ANSI Strict	"ANSI extensions you disable with ANSI Strict" on page 47
ANSI Keywords Only	"Additional keywords" on page 50
Expand Trigraphs	"Disabling trigraph characters" on page 49

This option…	Is described here…
Multi-Byte Aware	“Using multibyte strings and comments” on page 53
Direct to SOM	“Creating Direct-to-SOM Code” on page 105
Map Newlines to CR	“Using MPW C newlines” on page 76
Relaxed Pointer Type Rules	“Relaxing pointer checking” on page 56
Enums Always Int	“Enumerated constants of any size” on page 51
Use Unsigned Chars	“Chars always unsigned” on page 52

This is the C/C++ Warnings settings panel:

Figure 1.2 The C/C++ Warnings Settings Panel



This table describes where to find more information on its options:

This option…	Is described here…
Treat All Warnings As Errors	“Treat warnings as errors” on page 60
Illegal Pragmas	“Illegal pragmas” on page 60
Empty Declarations	“Empty declarations” on page 61
Possible Errors	“Possible unwanted side effects” on page 61

This option…	Is described here…
Unused Variables	"Unused variables" on page 62
Unused Arguments	"Unused arguments" on page 63
Extra Commas	"Extra commas" on page 64
Extended Error Checking	"Extended type checking" on page 65
Hidden virtual functions	"Function hiding" on page 66

What's New

This section describes the new documentation in this manual.

The long long type

Metrowerks C/C++ now has a 64-bit integer, the `long long`. See "64-bit integers" on page 46.

Turning off register coloring in the 68K compiler

You can now turn off register coloring in the 68K Mac OS compiler. This is useful when you're debugging code. See "Register coloring" on page 24.

More information on enumerated types

This manual now explains how the compiler implements enumerated types and on how to use enumerators that are large enough to be a `long long`. See "Enumerated types" on page 28.

New pragmas

There are three new pragmas:

- "longlong" on page 198
- "longlong_enums" on page 199
- "no_register_coloring (68K Macintosh and Magic Cap only)" on page 202

New intrinsic functions

There are three new PowerPC intrinsic functions, described on "Rotating the contents of a variable" on page 83.

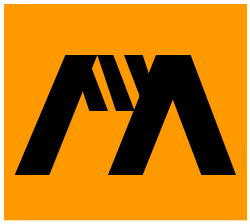
Improved documentation

There is new documentation on the MIPS and Win32/x86 inline assemblers. For more information, see "Overview of MIPS Assembler Notes" on page 161 and "Overview of Win32/x86 Assembler Notes" on page 167.

And there's improved documentation on these pragmas:

- "code_seg (Win32/x86 only)" on page 177
- "data_seg (Win32/x86 only)" on page 182
- "init_seg (Win32/x86 only)" on page 196
- "inline_depth (Win32/x86 only)" on page 197
- "warning (Win32/x86 only)" on page 225

2



C and C++ Language Notes

This chapter describes how Metrowerks handles the C programming language. Since many of the features in C are also in C++, this chapter is where you'll find basic information on C++ also.

Overview of C and C++ Language Notes

This chapter describes how Metrowerks handles the C programming language, and basic information on C++. For more information on the parts of the C++ language that are unique to C++, see "Overview of C++ Language Notes" on page 85.

In the margins of this chapter are references to K&R §A, which is Appendix A, "Reference Manual," of *The C Programming Language, Second Edition* (Prentice Hall) by Kernighan and Ritchie. These references show you where to look for more information on the topics discussed in the corresponding section.

This chapter contains the following sections:

- "The Metrowerks Implementation of C and C++" on page 20 explains how Metrowerks C and C++ implement certain parts of the standard.
- "Number Formats" on page 30 describes how C and C++ use store integers and floating-point numbers. This section has separate explanations for the 68K compiler and the PowerPC compiler.
- "Calling Conventions" on page 35 explains how C and C++ functions pass their arguments and return their values. This section has separate explanations for the 68K compiler and the PowerPC compiler.

- "Extensions to C or C++" on page 40 describe some of Metrowerks C and C++'s extensions to the C and C++ standards. You can disable most of these extensions with options in the C/C++ Language settings panel.
- "Warnings for Common Mistakes" on page 59 explains some options that check for common typographical mistakes. These options are in the C/C++ Warnings settings panel.
- "Generating Code for Specific 68K Processors (Macintosh Only)" on page 67 describes how to generate code optimized for the MC68020 and MC68881.
- "Calling MPW Functions" on page 72 describes how to use an MPW library in a CodeWarrior project.
- "Calling Macintosh Toolbox Functions (Macintosh Only)" on page 77 explains CodeWarrior's support for the Macintosh Toolbox.
- "Intrinsic PowerPC Functions (Macintosh Only)" on page 80 explains some functions that are built into Metrowerks C/C++ for PowerPC.

The Metrowerks Implementation of C and C++

This section describes how Metrowerks implements many parts of the C and C++ programming languages. For information on the parts of the C++ language that are specific to C++, see "Overview of C++ Language Notes" on page 85.

This section contains the following:

Identifiers

(K&R, §A2.3) The C and C++ compilers let you create identifiers of any size. However, only the first 255 characters are significant for internal and external linkage.

The C++ compiler creates mangled names in which all the characters in are significant. You do not need to keep your class and class member names artificially short to prevent the compiler from creating mangled names that are too long.

Include files

(K&R, §A12.4) The C and C++ compilers can nest #include files up to 32 times. An include file is nested if another #include file uses it in an `#include` statement. For example, if `Main.c` includes the file `MyFunctions.h`, which includes the file `MyUtilities.h`, the file `MyUtilities.h` is nested once.

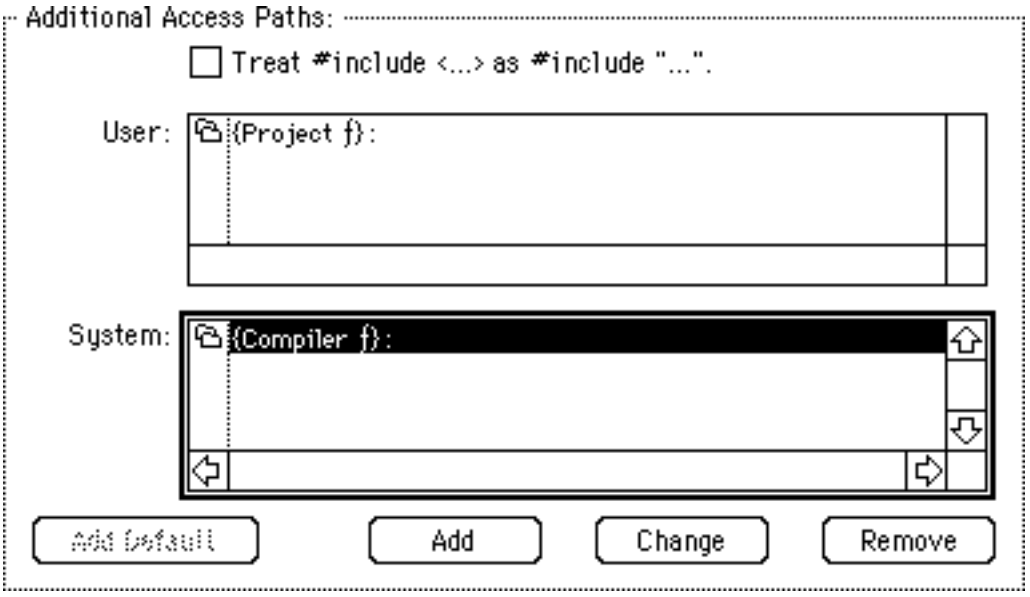
You can use full path names in `#include` directives, as in this example:

```
#include "HD:Tools:my headers:macros.h"
```

TIP: To add folders to the Access Paths settings panel, see the CodeWarrior IDE User's Guide.

The CodeWarrior IDE lets you specify where the compiler looks for #include files with the Access Paths settings panel, shown in Figure 2.1. It contains two lists of folders: the User list and the System list. By default, each list contains one folder. The User list contains `{Project` *`f`*`}`, which is the folder that the project file is in and all the folders it contains. The System list contains `{Compiler` *`f`*`}`, which is the folder that the compiler is in and all the folders it contains.

Figure 2.1 The Access Paths settings panel



The compiler searches for an #include file in either the System list or both the User and System lists, depending on which characters enclose the file. If you enclose the file in brackets (`#include <stdio.h>`), the compiler looks for the file in the System lists' folders section. If you enclose the file in quotes (`#include "myfuncs.h"`), the compiler looks for the file in the User list's folders and then in the System list's folders. In general, use brackets for include files that are for a large variety of projects and use quotes for include files that are for a specific project.

TIP: If you're using the compilers under MPW, you can specify where to find #include files with the `-i` compiler option and the `{CIncludes}` variable, described in Command-Line Tools Manual and MPW Command Reference.

The sizeof() operator

The `sizeof()` operator returns a number of type `size_t`, which this compiler defines to be `unsigned long int` (in `stddef.h`). If

your code assumes that `sizeof()` returns a number of type `int`, it may not work correctly.

Register variables

(K&R, §A4.1, §A8.1) The C and C++ compilers automatically allocate local variables and parameters to registers according to how frequently they're used and how many registers are available. If you're optimizing for speed, the compilers give preference to variables used in loops.

The PowerPC and 68K Macintosh compilers give preference to variables declared to be `register`, but do not automatically assign them to registers. For example, the compilers are more likely to place a variable from an inner loop in a register than a variable declared register.

The Win32/x86 compiler ignores the `register` declaration and decides on its own which variables to place in registers.

The PowerPC Macintosh compiler can use these registers for local variables:

- `GPR13` through `GPR31` for integers and pointers
- `FPR14` through `FPR31` for floating point variables.

The 68K Macintosh and Magic Cap compilers can use these registers for local variables:

- `A2` through `A4` for pointers
- `D3` through `D7` for integers and pointers.

If you turn on the **68881 Codegen** option, the 68K compilers also use these registers:

- `FP4` through `FP7` for 96-bit floating-point numbers

The Win32/x86 compiler can use these registers for local variables:

- `EAX`
- `EBX`
- `ECX`

- `EDX`
- `ESI`
- `EDI`

Register coloring

The Macintosh and Magic Cap compilers can also perform an additional register optimization, called *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're using `i` and `j` at the same time and place them in different registers:

```
int k = i + j;
```

To let the PowerPC compiler perform register coloring, turn on the **Global Optimizer** option in the PPC Processor settings panel and set the Level to 1 or more. To let the 68K Macintosh and Magic Cap compilers perform register coloring, turn on the **Global Register Allocation** option in the 68K Processor settings panel. The **Global Optimizer** option corresponds to the `global_optimizer` pragma, described on "global_optimizer, optimization_level (PowerPC Macintosh only)" on page 192. The **Global Register Allocation** option corresponds to the `no_register_coloring` pragma, described on "no_register_coloring (68K Macintosh and Magic Cap only)" on page 202.

If register coloring is on while you debug your project, it may appear as though there's something 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, turn off register coloring or declare the variables you want to watch as volatile.

Volatile variables

(K&R, §A4.4) When you declare a variable to be volatile, both the C or C++ compilers take the following precautions:

- It does not store the variable in a register.
- It computes the variable's address every time a piece of code references the variable.

Listing 2.1 shows an example of volatile variables.

Listing 2.1 volatile variables

```
void main(void)
{
  int i[100];
  volatile int a, b;

  a = 5;
  b = 20;

  i[a + b] = 15;
  i[a + b] = 30;
}
```

The compiler does not place the value of `a`, `b`, or `a+b` in register. Also, the compiler re-calculates a+b in both assignment statements.

Limits on variable sizes

(K&R, §A4.3, §A8.3, §A8.6.2) The Macintosh and Magic Cap C/C++ compilers let you declare structs and arrays to be any size, but place some limits on how you allocate space for them:

- A function cannot contain more than 32K of local variables. To avoid this problem, do one of the following:
  - Dynamically allocate large variables.
  - Declare large variables to be `static`. Note that if you're using a 68K compiler, you may run into the 32K limit on global variables, described below.
- If you're using a 68K compiler, you cannot declare a global variable that is over 32K unless you use far data. You must do one of the following:
  - Dynamically allocate the variable.
  - Use the `far` qualifier when declaring the variable.
  - Turn on the **Far Data** option in the Processor settings panel or use the pragma `far_data`.

The example below shows how to declare a large struct or array.

```
int i[50000];     // USUALLY OK.
                  // Wrong only when you use
                  // 68K compiler and turn off
                  // the Far Data option in the
                  // Processor settings panel

far int j[50000]; // ALWAYS OK.

int *k;
&k = malloc(50000 * sizeof(int));
                  // ALWAYS OK.
```

- Bitfields can be only 32 bits or less.

The Win32/x86 compiler places no limits on how large variables can be or how you allocate them.

Declaration specifiers

CodeWarrior lets you choose how to implement a function or variable with the declaration specifier `__declspec(`*arg*`)`, where *arg* specifies how to implement it. The Macintosh and Win32/x86 have different sets of arguments

For 68K and PowerPC Macintosh code, *arg* can be one of the following values:

- `__declspec(internal)` lets you specify that this variable or function is internal and not imported. It corresponds to the pragma `internal`, described at "internal (Macintosh only)" on page 197.
- `__declspec(import)` lets you import this variable or function which is in another code fragment or shared library. It corresponds to the pragma `import`, described at "import (Macintosh only)" on page 195.
- `__declspec(export)` lets you export this variable or function from this code fragment or shared library. It corresponds to the pragma `export`, described at "export (Macintosh only)" on page 186.
- `__declspec(lib_export)` ignores the pragmas export, import, and internal for this variable or function. It corresponds to the pragma `lib_export`, described at "lib_export (Macintosh only)" on page 198.

For Win32/x86 code, *arg* can be one of the following values:

- `__declspec(dllexport)` specifies that this function or variable is exported from the executable or DLL that defines it.
- `__declspec(dllimport)` specifies that this function or variable is imported from another DLL or executable.
- `__declspec(naked)` specifies that this function is entirely implemented with assembler code and the compiler does not need to produce any prefix or suffix code. It's the same as using the `asm` keyword.
- `__declspec(thread)` specifies that a copy of this global variable (i.e. `static` or `extern`) is created for each separate thread in this program. Creating separate copies can simplify

multi-threaded applications, since this is a reentrant way to refer to global storage. Note these restrictions on `__declspec(thread)`:

- You cannot use it in a DLL that's dynamically loaded (that is, your program specifically makes a runtime request for the DLL). You can use it in DLLs that are statically linked to your application and are implicitly loaded when your application is launched.
- If you declare a variable as `__declspec(thread)`, you cannot use its address as an initializer, since the program can determine the address only at run-time.

Enumerated types

This section describes how the C/C++ selects the underlying integer type for an enumerated type. There are two different different strategies, depending on the setting of the **Enum Always Int** option in the C/C++ Language settings panel, which corresponds to the `enumsalwaysint` pragma.

If **Enums Always Int** is on, the underlying type is always `signed int`. All enumerators must be no larger than a `signed int`. However, if the **ANSI Strict** option is off, enumerators that can be represented as an `unsigned int` are implicitly converted to `signed int`. (The **ANSI Strict** option is in the C/C++ Language settings panel and corresponds to the `ANSI_strict` pragma.)

Listing 2.2 Turning on the Enums Always Int option

```
#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 **Enums Always Int** is off, the compiler picks one of the following:

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

Listing 2.3 Turning off the Enums Always Int option

```
#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 will only use `long long` base types if is the `longlong_enums` pragma is on. (There is no settings panel option corresponding to the `longlong_enums` pragma)

Listing 2.4 Turning on longlong_enums pragma

```
#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 { d=-1,e=0x80000000 };   // OK: base type: signed long long
```

When the longlong_enums pragma is off and **ANSI strict** is on, you cannot mix huge unsigned 32-bit enumerators (greater than `0x7FFFFFFF`) and negative enumerators. If both the `longlong_enums` pragma and the **ANSI strict** option are off, huge unsigned 32-bit enumerators are implicitly converted to signed 32-bit types.

Listing 2.5 Turning off the longlong_enums pragma

```
#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)
```

For more information on **Enums Always Int**, see "Enumerated constants of any size" on page 51. For more information on **ANSI Strict**, see "ANSI extensions you disable with ANSI Strict" on page 47. For more information on the longlong_enums pragma, see "longlong_enums" on page 199.

Number Formats

(K&R, §A4.2) This section describes how the C and C++ compilers implement integer and floating-point types. You can also read `limits.h` for more information on integer types and `float.h` for more information on floating-point types.

This section contains the following:

68K Macintosh integer formats

The 68K Macintosh compiler lets you choose the size of an int with the **4-Byte Int** option in the Processor settings panel. In general, you'll turn this option on since it's easier to port your code to the

PowerPC compiler, which always uses 4-byte ints. However, 2-byte ints are slightly more efficient on the 68K, so you may want to turn this option off when efficiency is more important.

Table 2.1 shows the size and range of the integer types for a 68K compiler.

Table 2.1 68K Macintosh integer types

For this type	If this is true…	Size is	and its range is
`bool`	Always true	8 bits	`true` or `false`
`char`	Use Unsigned Chars is off	8 bits	`-128` to `127`
	Use Unsigned Chars is on	8 bits	`0` to `255`
`signed char`	Always true	8 bits	`-128` to `127`
`unsigned char`	Always true	8 bits	`0` to `255`
`short`	Always true	16 bits	`-32,768` to `32,767`
`unsigned short`	Always true	16 bits	`0` to `65,535`
`int`	4-Byte Ints is off	16 bits	`-32,768` to `32,767`
	4-Byte Ints is on	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned int`	4-Byte Ints is off	16 bits	`0` to `65,535`
	4-Byte Ints is on	32 bits	`0` to `4,294,967,295`
`long`	Always true	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned long`	Always true	32 bits	`0` to `4,294,967,295`
`long long`	Always true	64 bits	`-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`
`unsigned long long`	Always true	64 bits	`0` to `18,446,744,073,709,551,615`

68K Macintosh floating-point formats

You can choose the size of a double with the **8-Byte Doubles** option. In general, turn this option off since 8-byte (or 64-bit) doubles are less efficient than others. However, if you are porting code that relies on 8-byte doubles, turn this option on.

You can also choose to create code that is optimized for machines with a 68040 processor or a 68881 floating point unit. If you turn on the **68881 Codegen** option in the Processor settings panel, the compiler uses floating-point operations and types that are designed specifically for those chips. If you create code with the **68881 Codegen** option on and try to run it on a machine that does not have a 68040 or 68881, the code will crash. Turn on the **68881 Codegen** option only if the code contains lots of floating-point operations, must be as fast as possible, and you're sure the code will be used only on machines that contain a 68040 or 68881. Table 2.2 shows the size and range of the floating-point types for a 68K compiler.

Table 2.2 68K Macintosh floating point types

For this type	If this is true…	Its size is	and its range is
`float`	Always true	32 bits	`1.17549e-38` to `3.40282e+38`
`short double`	Always true	64 bits	`2.22507e-308` to `1.79769e+308`
`double`	8-Byte Doubles is on	64 bits	`2.22507e-308` to `1.79769e+308`
	8-Byte Doubles is off and 68881 Codegen is off	80 bits	`1.68105e-4932` to `1.18973e+4932`
	8-Byte Doubles is off and 68881 Codegen is on	96 bits	`1.68105e-4932` to `1.18973e+4932`
`long double`	68881 Codegen is off	80 bits	`1.68105e-4932` to `1.18973e+4932`
	68881 Codegen is on	96 bits	`1.68105e-4932` to `1.18973e+4932`

PowerPC Macintosh, Magic Cap, and Win32/x86 integer formats

The PowerPC Macintosh, Magic Cap, and Win32/x86 compilers do not let you change the sizes of integers. The size of a `short int` is always 2 bytes and the size of `int` or `long int` is always 4 bytes.

Table 2.3 shows the size and range of the integer types for the PowerPC Macintosh, Magic Cap, and Win32/x86 compilers.

Table 2.3 PowerPC, Magic Cap, and Win32/x86 Integer Types

For this type	If this is true…	Size is	and its range is
`bool`	Always true	8 bits	`true` or `false`
`char`	Use Unsigned Chars is off	8 bits	`-128` to `127`
	Use Unsigned Chars is on	8 bits	`0` to `255`
`signed char`	Always true	8 bits	`-128` to `127`
`unsigned char`	Always true	8 bits	`0` to `255`
`short`	Always true	16 bits	`-32,768` to `32,767`
`unsigned short`	Always true	16 bits	`0` to `65,535`
`int`	Always true	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned int`	Always true	32 bits	`0` to `4,294,967,295`
`long`	Always true	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned long`	Always true	32 bits	`0` to `4,294,967,295`
`long long`	Always true	64 bits	`-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`
`unsigned long long`	Always true	64 bits	`0` to `18,446,744,073,709,551,615`

WARNING! Do not turn off the **4-Byte Ints** option in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's off. It is on by default.

PowerPC Macintosh and Win32/x86 floating-point formats

Table 2.4 shows the sizes and ranges of the floating point types for the PowerPC Macintosh and Win32/x86 compilers.

Table 2.4 PowerPC Macintosh and Win32/x86 floating point types

Type	Size	Range
`float`	32 bits	`1.17549e-38` to `3.40282e+38`
`short double`	64 bits	`2.22507e-308` to `1.79769e+308`
`double`	64 bits	`2.22507e-308` to `1.79769e+308`
`long double`	64 bits	`2.22507e-308` to `1.79769e+308`

Magic Cap Floating-Point Formats

Table 2.5 shows the size and range of the floating-point types for the Magic Cap compiler.

Table 2.5 Magic Cap floating point types

Type	Size	Range
`float`	32 bits	`1.17549e-38` to `3.40282e+38`
`short double`	64 bits	`2.22507e-308` to `1.79769e+308`
double	96 bits	`1.68105e-4932` to `1.18973e+4932`
`long double`	96 bits	`1.68105e-4932` to `1.18973e+4932`

WARNING! Do not turn on the **8-Byte Doubles** option in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's on. It is off by default.

Calling Conventions

(K&R, §A8.6.3) This section describes the C and C++ calling conventions for both the Macintosh, Magic Cap, and Win32/x86 compilers. It contains the following:

68K Macintosh calling conventions

The 68K Macintosh and Magic Cap compilers pass all parameters on the stack in reverse order. This list describes where the compiler places a return value:

- It returns an integer values in register `D0`.
- It returns a pointer value in register `A0`.
- If it returns a value of any other type, the caller reserves temporary storage area for that type in the caller's stack and passes a pointer to that area as the last argument. The callee returns its value in the temporary storage area.

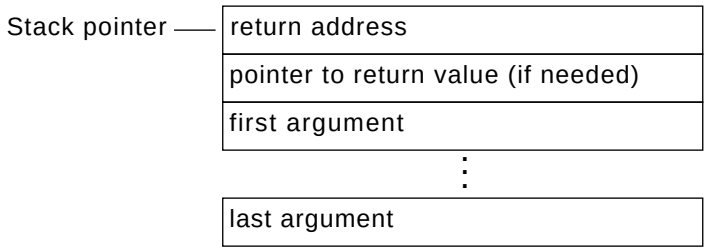
There are two options which can change how the compiler returns a value:

- If you turn on either the pragma `pointers_in_D0` or pragma `mpwc`, the compiler returns pointer values in register `D0`. Use one of these pragmas if you're calling a function declared in an MPW library. For more information, see "Calling MPW Functions" on page 72.

- If the **68881 Codegen** option is on, the compiler returns 96-bit floating-point values in register FP0.

Figure 2.2 shows what the stack looks like when you call a C function with the 68K Macintosh and Magic Cap compiler.

Figure 2.2 Calling a C function



PowerPC calling conventions

The consortium behind the PowerPC dictates a standard set of calling conventions that Metrowerks C/C++ for PowerPC follows. For more information on these calling conventions, see *Inside Macintosh: PowerPC System Software*. The rest of this section describes how Metrowerks C/C++ implements these standards.

The compiler reserves space for a function's parameters in two places: it reserves space for all parameter values in a structure in the caller's parameter area, and then it copies as many parameters as possible in registers. If the compiler copies a parameter into a register, it does not also copy it onto the parameter stack, but the compiler still reserves space for it on the stack. Placing parameters in registers avoids memory references to the parameter area and speeds up your programs.

NOTE: A word is eight bytes on the PowerPC.

In the parameter area, parameters are laid out in the order they appear, with the left-most parameter at the lowest offset. Each parameter starts at a word boundary regardless of size. For example, characters take up a word and doubles may not be on a double

word boundary. Signed integers smaller than a word are sign-byte extended to a word. Unsigned chars are zero-extended.

In the registers, the compiler maps the first eight words of the parameter area — excluding floating point values — to the general purpose registers `r3` through `r10`. Integers and pointers take up one register each. Composite parameters (such as structs, classes, and arrays) take up as many consecutive registers as they need. Note that the compiler maps composite parameters are to the registers as raw data, not as individual members or elements. For example, an array of six chars uses two registers: all of the first and the top half of the second.

NOTE: A composite parameter may be both in registers and the parameter stack. If the parameter starts in or before the eighth word and ends after the eighth word, the compiler stores part of it in registers and the rest on the parameter stack.

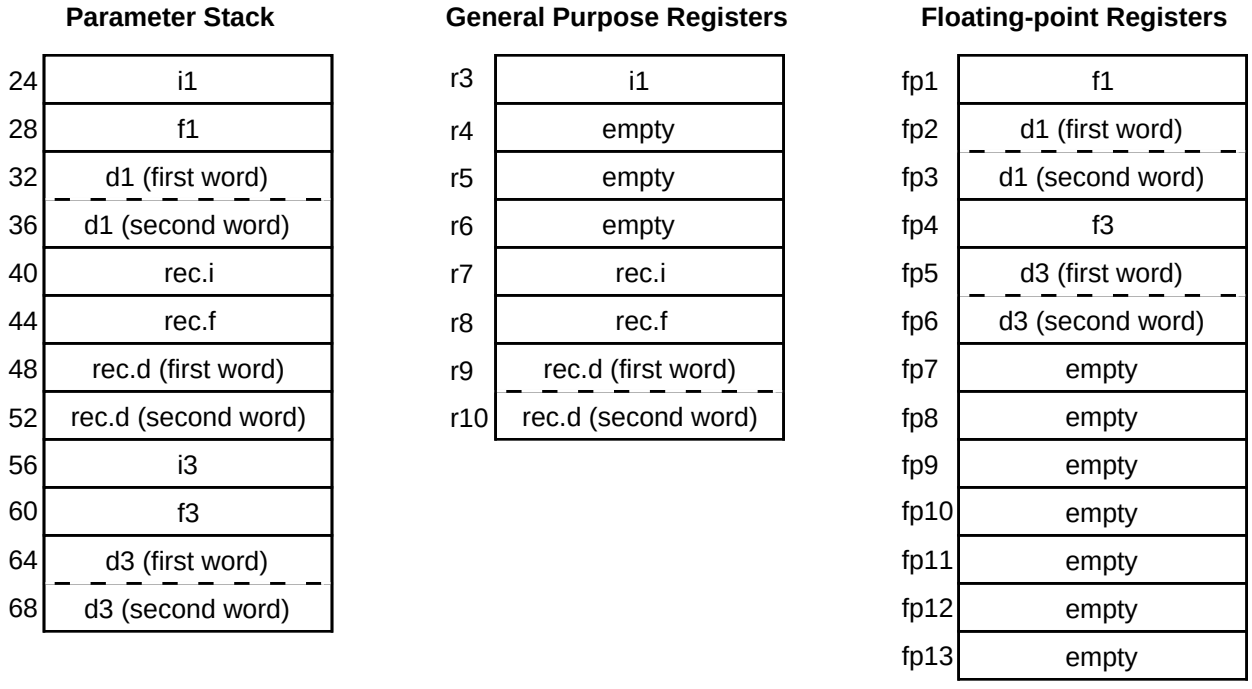
Floating-point values are mapped to the floating point registers `fp1` through `fp13`. The compiler maps only free variables and not floating-point values contained in composite types. If the floating-point parameter appears within the first eight words, the compiler does not use the corresponding general register or pair of registers. The compiler does not use the register but simply skips it. The compiler does not skip floating-point registers but uses them consecutively.

If a function does not have a prototype or has a variable argument list, the compiler copies the floating-point arguments into both general purpose registers and floating-point registers. In other words, the general purpose registers contain the first eight bytes of *all* parameters, and the floating-point registers contain duplicates of the floating-point parameters. The compiler performs this duplication since the function may be expecting either floats or raw data. If the function definition specifies floats, it will look for the parameters in the floating-point registers. If the function accepts anything and interprets the data itself (like `printf()`), it will look for the parameters in the general purpose registers.

Figure 2.3 shows how the compiler would store the parameters in function `foo()`, shown below. Note that `r4`, `r5`, and `r6` are empty and that the floating-point members of the struct are not stored in floating-point registers. Also, the compiler fills up the floating-point registers one after the other, even though the floating-point parameters do not follow each other.

```
typedef struct rec {
  int i;
  float f;
  double d;
} rec;

void foo( int i1, float f1, double d1, rec r,
          int i3, float f3, double d3 );
```

Figure 2.3 PowerPC parameter passing example



This list describes where the compiler places a return value:

- It returns integer values in `r3`.
- It returns float and double floating-point values in `fp1`.
- If it returns a composite type (such as a struct, class, or array), it allocates area for the return value in a temporary storage area, and returns a pointer to that area as an implicit left-most parameter (that is, in `r3`).

Magic Cap calling conventions

The Magic Cap compiler uses the same calling conventions as the 68K Macintosh compiler with the **MPW C Calling Conventions** option on and the `d0_pointers` pragma on. For more information, see "68K Macintosh calling conventions" on page 35, "Declaring MPW C functions (Macintosh Only)" on page 75, and "d0_pointers (68K Macintosh only)" on page 180.

Win32/x86 calling conventions

The Win32/x86 C/C++ compiler lets you choose how it calls functions with these types of declaration: `__stdcall` and `__thiscall`.

If you don't use a declaration specifier, the compiler uses the default calling convention. It pushes all parameters onto the stack in right to left order, so the first parameter in the list is on top of stack when the call is made. It expands each parameter to at least 32 bits on the stack and pads structs to an even number of 32 bit longwords. The caller removes the parameters from the stack. The compiler returns the function's value in one of these ways:

- It returns integer and pointer values in the EAX register.
- It returns floating point values on the floating point processor stack
- It returns structures and classes by passing an additional parameter with the address of a temporary variable and pushes that address onto the stack after all explicit parameters.

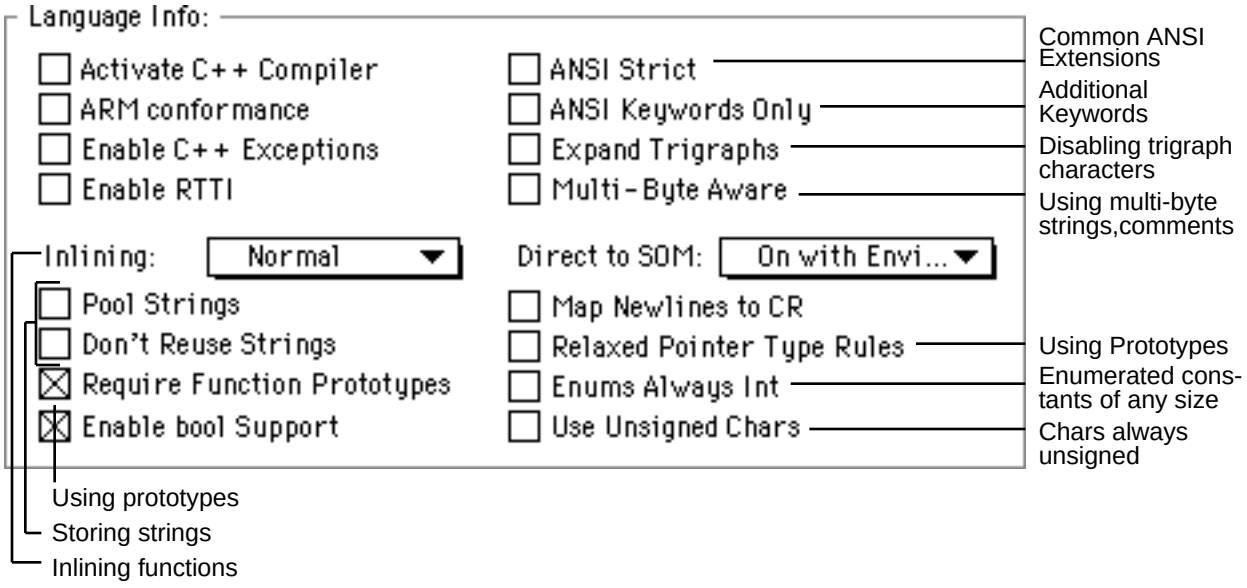
If you're declaring a function for an API, specify the standard calling convention with `__stdcall`. It's the same as the default calling convention, except that the callee removes parameters from stack.

If you're declaring a non-static member function, the compiler automatically uses the `__thiscall` calling convention unless you explicitly specify the standard calling convention with `__stdcall`. The `__thiscall` calling convention is the same as the standard calling convention, except that it passes the `this` pointer in the `ECX` register.

Extensions to C or C++

This section describes some of Metrowerks C and C++'s extensions to the C and C++ standards. You can disable most of these extensions with options in the Language preference panel, as shown in Figure 2.4.

Figure 2.4 Setting C Options in the C/C++ Languages Settings Panel



NOTE: For more information on the options in the upper right corner of the dialog (**Activate C++ Compiler**, **ARM Conformance**, **Enable Exception Handling**, **Don't Inline**, and **Enable RTTI**), as well as **Enable bool support** and **Direct to SOM** see "Overview of C++ Language Notes" on page 85. For more information on enable bool support, see "Using the bool type" on page 96. For more information on Map Newlines to CR, see "Using MPW C newlines" on page 76.

These are the extensions described in this section:

- "ANSI extensions you can't disable" on page 42 describes extensions you cannot disable. These extensions are common to many compilers, especially Macintosh compilers.
- "ANSI extensions you disable with ANSI Strict" on page 47 describes extensions you can disable with the **ANSI Strict** option. These extensions are common to many compilers.
- "Disabling trigraph characters" on page 49 describes how to prevent the compiler from expanding trigraph characters. You can disable this extension with the **Expand Trigraphs** option.
- "Additional keywords" on page 50 describes three additional words that the compiler recognizes as keywords. You can disable this extension with the **ANSI Keywords Only** option.
- "Enumerated constants of any size" on page 51 describes how Metrowerks C and C++ create enumerated constants of any size. You can disable this extension with the **Enums Always Int** option.
- "Chars always unsigned" on page 52 describes how Metrowerks C and C++ lets you treat a `char` declaration as an `unsigned char` declaration. You can enable this extension with the **Use Unsigned Chars** option.
- "Inlining functions" on page 52 describes how to choose the way in which Metrowerks C and C++ inline your functions. You choose with the **Inlining** menu.
- "Using multibyte strings and comments" on page 53 describes how to use multibyte strings and comments (such as

Kanji). You can enable this extension with the **Multi-Byte Aware** option.

- "Using prototypes" on page 54 describes how to control how strictly Metrowerks C and C++ enforce prototypes. There are two options and a pragma that control prototypes: the **Require Function Prototypes** option, the **Relaxed Pointer Type Rules** option, and the pragma `ignore_oldstyle`.
- "Storing strings (Macintosh only)" on page 56 describes how to control how to store strings. There are two options that control strings: **Pool Strings** and **Don't Reuse Strings**.

ANSI extensions you can't disable

This section describes some extensions to the ANSI C and C++ standards that you cannot disable with any option in the project settings. Many compilers, especially Macintosh compilers, support these extensions.

These extensions are as follows:

- "Multibyte characters (Macintosh Only)" on page 43
- "Declaring variables by address (Macintosh Only)" on page 43
- "Opcode inline functions (68K Macintosh Only)" on page 43
- "Specifying the registers for arguments (68K Macintosh Only)" on page 45
- "64-bit integers" on page 46

Multibyte characters (Macintosh Only)

(K&R, §A2.5.2) The C and C++ compilers let you use multibyte character constants which contain 2 to 4 characters. Here are some examples:

Table 2.6 Multibyte character constant

Character constant	Equivalent hexadecimal
`'ABCD'`	`0x41424344`
`'ABC'`	`0x00414243`
`'AB'`	`0x00004142`

Declaring variables by address (Macintosh Only)

(K&R, §A8.7) The C and C++ compilers let you specify the address that a variable refers to. For example, this definition defines `MemErr` to contain whatever is at the address `0x0220`:

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

the variable `MemErr` contains whatever is at the address `0x220`.

TIP: Avoid using this extension to refer to low-memory globals. To ensure that your programs are compatible with future versions of the Mac OS, use the functions defined in the `LowMem.h` header file.

Opcode inline functions (68K Macintosh Only)

(K&R, §A8.6.3, §A10.1) The 68K C and C++ compilers let you declare a function that specifies the opcodes that it contains. When you call an opcode inline function, the compiler replaces the function call with those opcodes. To define an opcode inline function, replace the function body with an equals sign and the opcode. If there's

more than one opcode, enclose them in brackets. Listing 2.6 shows two opcode inline functions.

Listing 2.6 Declaring an opcode inline function

```
pascal OSErr FSpCatMove(FSSpec *from,FSSpec *to)
            = { 0x303C,0x000C,0xAA52 };

pascal void LineTo(short h,short v) = 0xA891;
```

NOTE: Only the 68K Macintosh C and C++ compilers lets you use opcode inline function declarations. However, all the C++ compilers let you use C++ inline functions, declared with the `inline` keyword.

Inline data (68K Macintosh Only)

The 68K C and C++ compilers let you include simple inline data with the asm declaration. Use this syntax:

```
asm { constant, constant, . . . }
asm ( constant, constant, . . . )
```

A *constant* can be a numeric constant or a string literal.

For example, this function:

Listing 2.7 Inline data example

```
void foo()
{
  asm ( (short)0x4e71,(short)0x4e71 );
    // two 68K NOP instructions
```

```
  asm { 0x4e714e71,0x4e714e71 };
    // four 68K NOP instructions

  asm ((char)'C',(char)'o',(short)'de',"Warrior");
}
```

Produces assembly code that looks like this:

Listing 2.8 Assembly code from inline data

```
  LINK   A6, #$0000
  NOP
  NOP                       ; First two NOPs
  NOP
  NOP
  NOP
  NOP                       ; Next four NOPs
  DC.B    "CodeWarrior\0"
  UNLK    A6
  RTS
```

Specifying the registers for arguments (68K Macintosh Only)

(K&R, §A8.6.3, §A10.1) The 68K C and C++ compilers let you can specify which registers that a function uses for its parameters and the return value. The registers `D0-D2`, `A0-A1`, and `FP0-FP3` are available.

When you declare the function, specify the registers by using the `#pragma parameter` statement before the declaration. When you define the function, specify the registers right in the argument list.

This is the syntax for the `#pragma parameter`:

`#pragma parameter` *return-reg* *func-name*`(`*param-regs*`)`

The compiler passes the parameters for the function *func-name* in the registers specified in *param-regs* instead of the stack, and returns any

return value in the register *return-reg*. Both *return-reg* and *param-regs* are optional.

For example, Listing 2.9 shows the declaration and definition of a function, in which `a` is passed in `D0`, `p` is passed in `A1`, `x` is passed in `FP0` and `f` is passed on the stack.

Listing 2.9 Using registers with functions

```
#pragma parameter __D2 function(__D0,__A1,__FP0)
short function(long a, Ptr p, long double x,
               short f);

short function(long a:__D0, Ptr p:__A1,
               long double x:__FP0, short f) :__D2
{
  // ...
}
```

64-bit integers

The C or C++ compiler lets you define a 64-bit integer with the type specifier `long long`. This is twice as large as a `long int`, which is a 32-bit integer. A `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`.

In an enumerated type, you can use an enumerator large enough for a `long long`. For more information, see "Enumerated types" on page 28. However, `long long` bitfields are not supported.

You can disable the `long long` type with the pragma `longlong`, described at "longlong" on page 198. There is no settings panel option to disable it. If this pragma is off, using `long long` causes a syntax error. To check whether this option is on, use `__option (longlong)`, described in "Options Checking" on page 229. By default, this pragma is on.

ANSI extensions you disable with ANSI Strict

This section describes some optional extensions to the ANSI C and C++ standards that you can enable by turning off the **ANSI Strict** option in the Language preference panel. Many compilers, including Metrowerks C and C++, support these extensions.

NOTE: You cannot compile most standard Macintosh applications if the **ANSI Strict** option is on. In general, use this option only if you have to check whether a program is strictly ANSI-conferment.

The optional ANSI extensions are the following. If you turn on the **ANSI Strict** option, the compiler generates an error if it encounters any of these extensions.

- "C++-style comments" on page 48
- "Unnamed arguments in function definitions" on page 48
- "A # not followed by argument in macro definition" on page 48
- "An identifier after #endif" on page 48
- "Using typecasted pointers as lvalues" on page 49

For more information on how this option affects enumerated types, see "Enumerated types" on page 28.

The **ANSI Strict** option corresponds to the pragma `ANSI_strict`, described at "ANSI_strict" on page 173. To check whether this option is on, use `__option (ANSI_strict)`, described at "ANSI_strict" on page 230. By default, this option is off.

C++-style comments

(K&R, §A2.2) In the C compiler, you can use C++-style comments. Anything that follows `//` on a line is considered a comment. For example:

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

Unnamed arguments in function definitions

(K&R, §A10.1) The C compiler lets you use an unnamed argument in a function definitions. For example:

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

A # not followed by argument in macro definition

(K&R, §A12.3) The C and C++ compilers do not generate an error if you use the quote token (#) in a macro definition and a macro argument does not follow it. For example:

```
#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"
```

An identifier after #endif

(K&R, §A12.5) The C and C++ compilers let you place an identifier token after `#endif` and `#else`. This extension helps you match an `#endif` statement with its corresponding `#if`, `#ifdef`, or `#ifndef` statement, as shown below:

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

If you turn on the **ANSI Strict** option, you can make the identifiers into comments, like this:

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

Using typecasted pointers as lvalues

The C and C++ compilers let you use a pointer that you've typecasted to another pointer type as an lvalue. For example:

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

Disabling trigraph characters

(K&R, §A12.1) The C and C++ compilers let you ignore trigraph characters. Many common Macintosh character constants look like trigraph sequences, and this extension lets you use them without including escape characters.

If you're writing code that must follow the ANSI standard strictly, turn on the **Expand Trigraphs** option in the Language preference panel. Be careful when you initialize strings or multi-character constants that contain question marks. For example:

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

The **Expand Trigraphs** option corresponds to the pragma `trigraphs`, described at "trigraphs" on page 219. To check whether this option is on, use `__option (trigraphs)`, described at "trigraphs" on page 235. By default, this option is off.

Additional keywords

(K&R, §A2.4) If you're writing code that must follow the ANSI standard strictly, turn on the **ANSI Keywords Only** option in the Language preference panel. The compiler generates an error if it encounters any of the Metrowerks C/C++ additional keywords.

This sections contains the following:

The **ANSI Keywords Only** option corresponds to the pragma `only_std_keywords`, described at "only_std_keywords" on page 204. To check whether this option is on, use `__option (only_std_keywords)`, described at "only_std_keywords" on page 233. By default, this option is off.

Macintosh and Magic Cap keywords

The 68K Macintosh, PowerPC Macintosh, and Magic Cap C /C++ compilers recognize three additional reserved keywords.

- `asm` lets you compile a function's body with built-in assembler. For more information on how to use the built-in assembler, consult "Overview of 68K Assembler Notes" on page 115 and "Overview of PowerPC Assembler Notes" on page 125. (K&R, §A10.1)
- `far` (68K only) lets you declare a variable or a function to use the far mode addressing regardless of how you set the options **Far Data**, **Far Virtual Function Tables**, and **Far String Constants** in the Processor settings. For more information on the far mode, see the *CodeWarrior IDE User's Guide*. (K&R, §A8.1)

NOTE: The PowerPC compiler ignores the `far` qualifier but does not generate an error.

- `pascal` lets you declare a function that uses Pascal calling conventions. For information, see "Calling Macintosh Tool-

box Functions (Macintosh Only)" on page 77. (K&R, §A8.6.3, §A10.1)

- `inline` lets you declare a C function to be inline. It works the same as `inline` in C++. For more information, see "Inlining functions" on page 52.

Win32/x86 keywords

The Win32/x86 compiler recognizes these keywords:

- `__stdcall` specifies that this function uses the standard calling convention. For more information, see "Win32/x86 calling conventions" on page 39.
- `asm` specifies that this function is entirely implemented with assembler code and the compiler does not need to produce any prefix or suffix code.

The Win32/x86 compiler ignores the `pascal` keyword and raises an error for the `far` keyword.

Enumerated constants of any size

(K&R, §A8.4) When the **Enums Always Int** option is on, the C or 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. When the option is off, the compiler makes an enumerated type the size of any integral type. It chooses the integral type with the size that most closely matches the size of the largest enumerated constant. The type could be as small as a `char` or as large as a `long int`.

For example:

```
enum SmallNumber { One = 1, Two = 2 };
  /* If Enums Always Int is off, this type will
     be the same size as a char.
     If the option is on, this type will be
     the same size as an int. */
```

```
enum BigNumber
  { ThreeThousandMillion = 3000000000 };
  /* If Enuns Always Int is off, this type will
     be the same size as a long int.
     If this option is on, the compiler may
     generate an error. */
```

For more information on how enumerated types are implemented, see "Enumerated types" on page 28.

The **Enums Always Int** option corresponds to the pragma `enumsalwaysint`, described at "enumsalwaysints" on page 184. To check whether this option is on, use `__option (enumsalwaysint)`, described at "enumsalwaysint" on page 231. By default, this option is off.

Chars always unsigned

When the **Use Unsigned Chars** option is on, the C/C++ compiler treats a `char` declaration as if it were an `unsigned char` declaration.

NOTE: If you turn this option on, your code may not be compatible with libraries that were compiled with this option turned off. In particular, your code may not work with the ANSI libraries included with CodeWarrior.

The **Use Unsigned Chars** option corresponds to the pragma `unsigned_char`, described at "unsigned_char" on page 220. To check whether this option is on, use `__option (unsigned_char)`, described at "unsigned_char" on page 235. By default, this option is off.

Inlining functions

Metrowerks C/C++ gives you several different ways to inline both C and C++ functions. When you call an inline function, the caller inserts the function's code instead of a function call. Inlining functions

makes your programs faster (since the compiler executes the function's code immediately without a function call), but possibly larger (since the function's code may be repeated in several different places).

If you turn off the **ANSI Keywords Only** option, you can declare C functions to be `inline`, just as you do in C++. And the **Inlining** menu lets you choose to inline all small functions, only functions declared inline, or no functions, as shown in the table below:

This option	Does this…
Don't Inline	Inlines no functions, not even C or C++ functions declared `inline`.
Normal	Inlines only C and C++ functions declared `inline` and member functions defined within a class declaration. Note that Metrowerks may not be able to inline all the functions you declare `inline`.
Auto-Inline	Lets the compiler choose which functions to inline. Also inlines C++ functions declared `inline` and member functions defined within a class declaration.

The **Don't Inline** option corresponds to the pragma `dont_inline`, described at "dont_inline" on page 183. To check whether this option is on, use `__option (dont_inline)`, described at "dont_inline" on page 231. By default, this option is off.

The **Auto-Inline** option corresponds to the pragma `auto_inline`, described at "auto_inline" on page 175. To check whether this option is on, use `__option (auto_inline)`, described at "auto_inline" on page 230. By default, this option is off.

Using multibyte strings and comments

To use multibyte strings or comments (such as Kanji), turn on the Multi-Byte Aware option. If you don't need multibyte strings or comments, turn this option off, since it slows down the compiler.

Using prototypes

(K&R, §A8.6.3, §A10.1) The C and C++ compilers let you choose how to enforce function prototypes:

- "Requiring prototypes" on page 54 explains the **Require Prototypes** option which forces you to prototype every function so you can find errors caused by forgotten prototypes.
- "Relaxing pointer checking" on page 56 explains the **Relaxed Pointer Type Rules** option which treats `char*`, `unsigned char*`, and `Ptr` as the same type.

Requiring prototypes

When the **Require Prototypes** option is on, the compiler generates an error if you use a function that does not have a prototype. This option helps you prevent errors that happen when you use a function before you define it. If you do not use function prototypes, your code may not work as you expect even though it compiles without error.

In Listing 2.10, `PrintNum()` is called with an integer argument but is later defined to take a floating-point argument.

Listing 2.10 Unnoticed type-mismatch

```
#include <stdio.h>

void main(void)
{
  PrintNum(1);       // NO: PrintNum() tries to
}                    // interpret the integer as a
                     // float. Prints 0.000000.
void PrintNum(float x)
{
  printf("%f\n", x);
}
```

When you run it, 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()` is not prototyped, the compiler does not know it needs 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 2.11, the compiler converts its argument to a floating-point number, and the function prints what you wanted.

Listing 2.11 Using a prototype to avoid type-mismatch

```
#include <stdio.h>

void PrintNum(float x); // Function prototype.

void main(void)
{
  PrintNum(1);          // OK: Compiler knows to
}                       // convert integer to float.
                        // Prints 1.000000.
void PrintNum(float x)
{
  printf("%f\n", x);
}
```

In other situations where automatic typecasting is not available, the compiler generates an error when an argument does not match the expected data type. Such a mismatched data type error is easy to locate at compile time. If you do not use prototypes, you get no error and the cause of the resulting unintentional behavior can be extremely difficult to track down.

The **Require Prototypes** option corresponds to the pragma `require_prototypes`, described at "require_prototypes" on page 211. To check whether this option is on, use `__option (require_prototypes)`, described at "require_prototypes" on page 234. By default, this option is on.

Relaxing pointer checking

When you turn on the **Relaxed Pointer Type Rules** option in the C/C++ Language settings panel, the compiler treats `char*`, `unsigned char*`, and `Ptr` as the same type. This option is especially useful if you're using code written before the ANSI C standard. This old code frequently used these types interchangeably. When compiling C++ code, the compiler ignores the setting of this option and always treats the types as different types.

The **Relaxed Pointer Type Rules** option corresponds to the pragma `mpwc_relax`, described at "mpwc_relax" on page 202. To check whether this option is on, use `__option (mpwc_relax)`, described at "mpwc_relax" on page 233.

Storing strings (Macintosh only)

The C and C++ compilers let you choose how to store strings:

- "Pooling strings" on page 56 describes the **Pool Strings** option which lets you save space in your program's TOC by collecting all your string constants into a single data object.
- "Using PC-relative strings" on page 57 describes the **PC-Relative Strings** option which lets you choose whether to store strings in your code resources or in your global data.
- "Reusing strings" on page 58 describes the **Don't Reuse Strings** option which lets you store only one copy of identical strings.

Pooling strings

If the **Pool Strings** option in the Language preference panel is on, the compiler collects all string constants into a single data object so your program needs one TOC entry for all of them. If this option is off, the compiler creates a unique data object and TOC entry for

each string constant. Turning this option on decreases the number of TOC entries in your program but increases your program's size, since it uses a less efficient method to store the string's address.

TIP: You can also change the size of the TOC with the **Store Static Data in TOC** option in the PPC Processor preference panel. For more information, see the CodeWarrior User's Guide.

This option is especially useful if your program is large and has many string constants or uses the Metrowerks Profiler.

NOTE: If you turn the **Pool Strings** option on, the compiler ignores the setting of the **PC-Relative Strings** option.

The **Pool Strings** option corresponds to the pragma `pool_strings`, described at "pool_strings" on page 208. To check whether this option is on, use `__option (pool_strings)`, described at "pool_strings" on page 234. By default, this option is off.

Using PC-relative strings

If the **PC-Relative Strings** option in the Processor preference panel is on, the compiler stores the string constants used in a local scope in the code segment and addresses these strings with PC-relative instructions. If this option is off, the compiler stores all string constants in the global data segment. This option helps keep your global data segment smaller.

NOTE: This option is available only with the 68K compilers. It is not available with the PowerPC compilers.

Regardless of how this option is set, the compiler stores string constants used in the global scope in the global data segment. For example:

```
#pragma pcrelstrings on

int f(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)
#pragma pcrelstrings reset
```

NOTE: If you turn the **Pool Strings** option on, the compiler ignores the setting of the **PC-Relative** Strings option.

The **PC-Relative Strings** option corresponds to the pragma `pcrelstrings`, described at "pcrelstrings (68K Macintosh only)" on page 206. To check whether this option is on, use `__option(pcrelstrings)`, described at "pcrelstrings (68K only)" on page 233. By default, this option is off.

WARNING! Do not turn off the **PC-Relative Strings** option in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's off. It is on by default.

Reusing strings

If the **Don't Reuse Strings** option in the C/C++ Languages settings panel is on, the compiler stores each string literal separately. If this option is off, the compiler stores only one copy of identical string lit-

erals. This option helps you save memory if your program contains lots of identical string literals which you do not modify.

For example, take this code segment:

```
char *str1="Hello";
char *str2="Hello"
*str2 = 'Y';
```

If this option is on, `str1` is `"Hello"` and `str2` is `"Yello"`. If this option is off, both `str1` and `str2` are `"Yello"`.

The **Don't Reuse Strings** option corresponds to the pragma `dont_reuse_strings`, described at "dont_reuse_strings" on page 184. To check whether this option is on, use `__option(dont_reuse_strings)`, described at "dont_reuse_strings" on page 231. By default, this option is on. (Strings are *not* reused.)

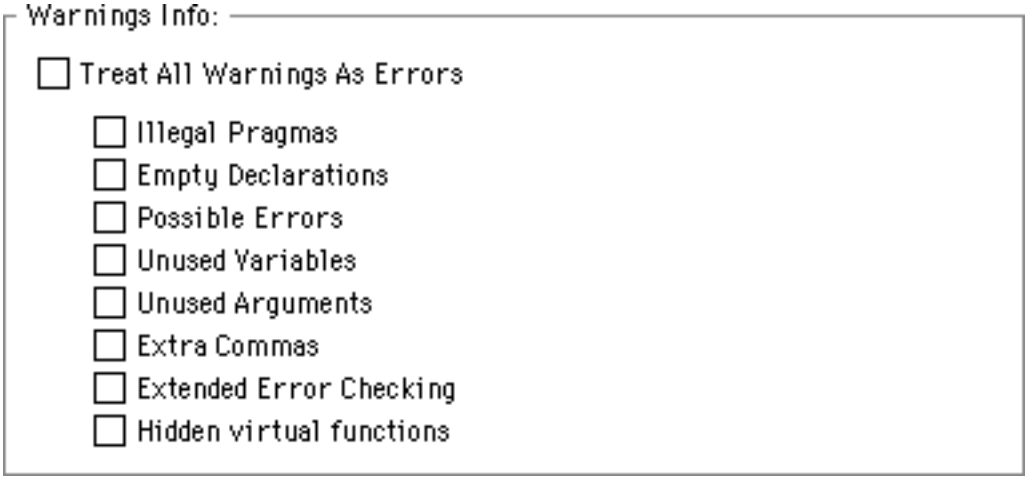
Warnings for Common Mistakes

This section describes the options in the Warnings preference panel, which check for common typographical mistakes.These mistakes are legal C and C++ code but might not do what you expect. When the compiler finds one of these possible mistakes, it generates a warning. Since these mistakes raise warnings, your code will compile and run even if the compiler finds one.

The options in this section warn you of the following:

The one option that isn't a warning is the **Treat All Warnings as Errors** option. If these option is on, the compiler treats all the warnings the compiler generates, including the ones described here, as errors, and it won't compile your code until you resolve them.

Figure 2.5 The C/C++ Warnings Settings Panel



Treat warnings as errors

When the **Treat All Warnings as Errors** option in the Warnings preference panel is on, the compiler treats all warnings as though they were errors. It will not compile a file until all warnings are resolved.

The **Treat All Warnings as Errors** option corresponds to the pragma `warning_errors`, described at"warning_errors" on page 221. To check whether this option is on, use `__option (warning_errors)`, described at "warning_errors" on page 236. By default, this option is off.

Illegal pragmas

If the **Illegal Pragmas** option is on, the compiler displays a warning when it encounters an illegal pragma. For example, these pragma statements generate warnings:

```
#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** option corresponds to the pragma `warn_illpragma`, described at "warn_illpragma" on page 223. To check whether this option is on, use `__option (warn_illpragma)`, described at "warn_illpragma" on page 235. By default, this option is off.

Empty declarations

If the **Empty Declarations** option is on, the compiler displays a warning when it encounters a declaration with no variables. For example:

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

The **Empty Declarations** option corresponds to the pragma `warn_emptydecl`, described at "warn_emptydecl" on page 221. To check whether this option is on, use `__option (warn_emptydecl)`, described at "warn_emptydecl" on page 235. By default, this option is off.

Possible unwanted side effects

If the **Possible Errors** option is on, the compiler checks for some common typographical mistakes that are legal C and C++ but that may have unwanted side effects, such as putting in unintended semicolons or confusing = and ==. The compiler generates a warning if it encounters one of these:

- An assignment in a logical expression or the condition in an `if`, `while`, or `for` expression. This check is useful if you frequently use = when you meant to use ==. For example:

```
if (a=b) f();          // WARNING: a=b is an
                       //          assignment
```

```
if ((a=b)!=0) f();    // OK: (a=b)!=0 is a
                      //     comparison

if (a==b) f();        // OK: (a==b) is a
                      //     comparison
```

- An equal comparison in a statement that contains a single expression. This check is useful if you frequently use == when you meant to use =. For example:

```
a == 0;         // WARNING: This is a comparison.
a = 0;          // OK: This is an assignment
```

- A semicolon (;) directly after a `while`, `if`, or `for` statement. For example, the statement generates an error 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 a comment. For example, these statements do not generate errors:

```
while (i++) ; // OK: White space separation
while (i++)   /*: Comment separation */ ;
```

The **Possible Errors** option corresponds to the pragma `warn_possunwant`, described at "warn_possunwant" on page 223. To check whether this option is on, use `__option (warn_possunwant)`, described at "warn_possunwant" on page 235. By default, this option is off.

Unused variables

If the **Unused Variables** option is on, the compiler generates a warning when it encounters a variable you declare but do not use. This check helps you find misspelled variable names and variables you have written out of your program. For example:

```
void foo(void)
{
  int temp, errer;       // ERROR: errer is
                         // misspelled
```

```
  error = do_something()
}    // WARNING: temp and error are unused.
```

If you need to declare a variable that you don't use, use the pragma `unused`, as in this example:

```
void foo(void)
{
  int i, temp, error;
#pragma unused (i, temp) /* Compiler won't warn
  error=do_something();   * that i and temp are
}                         * not used
                          */
```

The **Unused Variables** option corresponds to the pragma `warn_unusedvar`, described at "warn_unusedvar" on page 225. To check whether this option is on, use `__option (warn_unusedvar)`, described at "warn_unusedvar" on page 236. By default, this option is off.

Unused arguments

If the **Unused Arguments** option is on, the compiler generates a warning when it encounters an argument you declare but do not use. This check helps you find misspelled argument names and arguments you have written out of your program.

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

If you need to declare an argument that you don't use, there are two ways to avoid this warning. You can use the pragma `unused`, as in this example:

```
void foo(int temp, int error)
{
#pragma unused (temp)  /* Compiler won't warn
  error=do_something(); * that temp is not used
}                       */
```

You can also turn off the **ANSI Strict** option, and not give the unused argument a name, like this:

```
void foo(int /* temp */, int error)
{                       /* Compiler won't warn
#pragma unused (temp)    * that temp is not used
  error=do_something(); */
}
```

The **Unused Arguments** option corresponds to the pragma `warn_unusedarg`, described at "warn_unusedarg" on page 224. To check whether this option is on, use `__option (warn_unusedarg)`, described at "warn_unusedarg" on page 235. By default, this option is off.

Extra commas

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

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

The **Extra Commas** option corresponds to the pragma `warn_extracomma`, described at "warn_extracomma" on page 222. To check whether this option is on, use `__option (warn_extracomma)`, described at "warn_extracomma" on page 235. By default, this option is off.

Extended type checking

If the **Extended Error Checking** option is on, the C compiler generates a warning (not an error) if it encounters one of these syntax problems:

- A non-`void` function that does not contain a `return` statement. For example, this would generate a warning:

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

This would be OK:

```
void main()
{
  printf ("hello world\n");
}
```

- Assigning an integer or floating-point value to an `enum` type. For example:

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

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

NOTE: Both of these syntax problems are always errors in C++.

The C and C++ compilers generate a warning if it encounters this:

- An empty `return` statement (`return;`) in a function that is not declared `void`. For example, this code would generate a warning:

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err!=0)     return;
      // ERROR: Empty return statement
  // . . .
}
```

This would be OK:

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

  // . . .
}
```

The **Extended Error Checking** option corresponds to the pragma `extended_errorcheck`, described at "extended_errorcheck" on page 187. To check whether this option is on, use `__option (extended_errorcheck)`, described at "extended_errorcheck" on page 231. By default, this option is off.

Function hiding

If the **Hidden virtual functions** option is on, the compiler generates a warning if you declare a non-virtual member function that hides a virtual function in a superclass. One function hides another if it has the same name but a different argument types. For example:

```
class A {
  public:
    virtual void f(int);
    virtual void g(int);
};
```

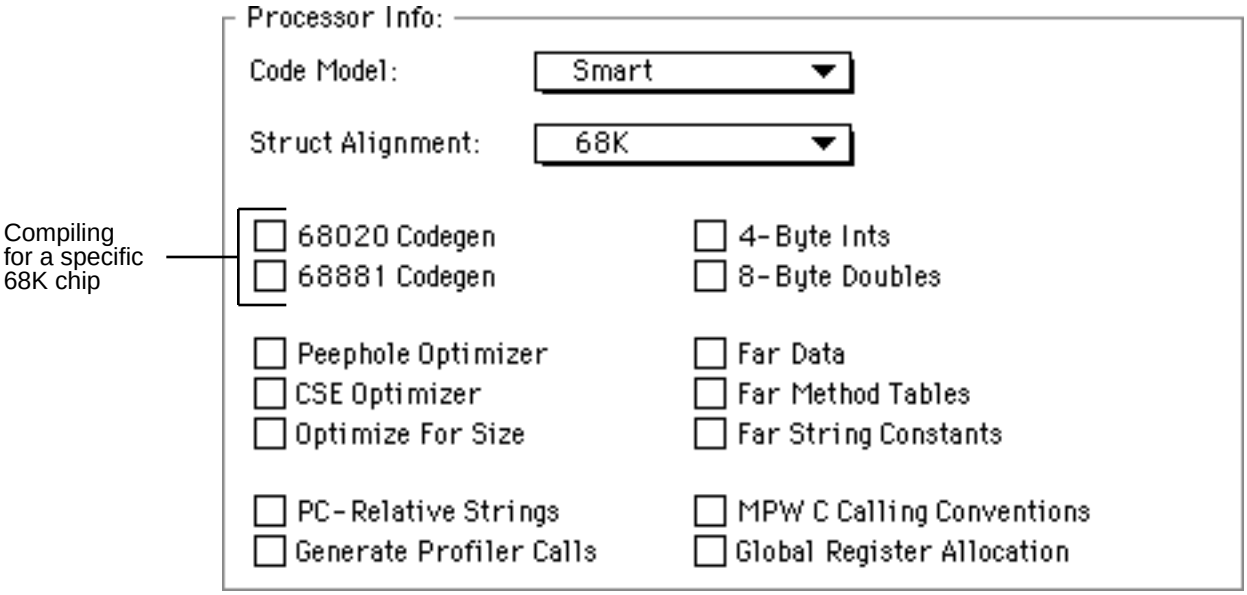
```
class B: public A {
  public:
    void f(char);        // WARNING:
                         //  Hides A::f(int)
    virtual void g(int); // OK:
                         //  Overrides A::g(int)
};
```

The **Hidden virtual functions** option corresponds to the pragma `warn_hidevirtual`, described at "warn_hidevirtual" on page 222. To check whether this option is on, use `__option(warn_hidevirtual)`, described at "warn_hidevirtual" on page 235. By default, this option is off.

Generating Code for Specific 68K Processors (Macintosh Only)

The CodeWarrior IDE lets you generate code for specific 68K processors: the MC68020 processor and the MC68881 floating-point unit. You can find these options in the Processor settings panel, shown in Figure 2.6.

Figure 2.6 Options to Generate Code for Specific 68K Processors



This sections contains the following:

TIP: Use these options only if your application will run solely on machines that have that processor and your application needs the extra efficiency that the processor provides. In general, if your application needs to be as fast as possible, compile it for the PowerPC. Most users who want fast applications have a Power Macintosh.

Metrowerks C and C++ let you compile different code depending on which processor you're compiling code for, with the `__option()` pre-processor function. Use `__option(code68881)` to check whether the **68881 Codegen** option is on. Use `__option(code68020)` to check whether the **68020 Codegen** op-

tion is on. The following example uses different code depending on whether the function is going to run on a machine with a MC68881:

```
int calc(double i)
{
#if __option (code68881)
  // Code optimized for the floating point unit.
#else
  // Code for any Macintosh
#endif
}
```

TIP: For more information on `__option()`, see “Options Checking” on page 229.

To check whether the computer on which your application is running has a specific processor use the `gestalt()` function. The following code sample displays an alert if the application is for an MC68881 and the machine does not have an MC68881:

```
void main(void)
{
#if __option (code68881)
  if (!HasFPU())    // Calls gestalt() to check
  {                 //  if the computer has FPU
    DisplayNoFPU(); // Displays an alert
    return;         //  saying there is no FPU
  }
#endif
  // . . .
}
```

TIP: For more information on `gestalt()`, see Chapter “Gestalt Manager” in Inside Macintosh: Operating System Utilities.

Note that `HasFPU()` and `DisplayNoFPU()` are not Toolbox functions. If you use this code, you must define these functions.

WARNING! Do not turn off the **68020 Codegen** and **68881 Codegen** options in Magic Cap code. Although the Magic Cap compiler lets you change the setting of these options, your code will not run correctly if they're off. They are on by default.

Generating code for the MC68020

The CodeWarrior IDE lets you take full advantage of the MC68020 processor. When you turn on the **68020 Codegen** option in the Processor preference panel, the C and C++ compilers use the extensions available in the MC68020 instruction set, including integer multiplication, integer division, and bit-field operations.

WARNING! Before your program runs code optimized for the MC68020, use the `gestalt()` function to make sure it is available. For more information on `gestalt()`, see Chapter "Gestalt Manager" in Inside Macintosh: Operating System Utilities.

Generating code for the MC68881

The CodeWarrior IDE lets you take full advantage the MC68881 floating-point unit. The MC68881 is built into most versions of the MC68040 processor, and it is included separately in many Macintosh computers that contain the MC68030 or MC68020 processor.

Metrowerks C and C++ give you two levels of support for the MC68881, as described below:

- No matter what you do, the Macintosh Toolbox uses the MC68881 for many floating-point functions.
- If you also turn on the **68881 Codegen** option in the Processor settings panel, the compiler generates code optimized for the MC68881 and stores variables declared `long double` or `extended` in 96 bits. It uses MC68881 instructions for basic arithmetic operations, such as addition, subtraction, multiplication, division, and comparisons. The header files `fp.h` and `math.h` use MC68881 instructions for many transcendental

and floating-point conversions. The compiled code is faster and computes the same results as code compiled with the option off.

Think carefully before you use the **68881 Codegen** option. Your code will not run on a Power Macintosh or any 68K Macintosh that does not have a MC68881. Even if you do not use the **68881 Codegen** option, the Macintosh toolbox will use the MC68881 to compute many floating-point functions.

WARNING! 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.

The rest of this section describes what happens when you turn on the **MC68881 Codegen** option.

Using the Extended data type

If you turn on the **68881 Codegen** option, the compiler stores any variable declared `extended` or `long double` in the Motorola 96-bit format, instead of the SANE 80-bit format. Both formats meet the IEEE standards for accuracy. The main difference between them is that the 96-bit format contains 16 bits of padding so that an extended number fits evenly into three 32-bit memory accesses.

`Types.h` defines the `extended` type. `SANE.h` contains two other type definitions: `extended80` and `extended96`. It also contains functions that convert between 80-bit and 96-bit formats: `x96tox80()` and `x80tox96()`.

NOTE: The PowerPC architecture does not support the `extended` type. Use `double` instead.

Using floating-point registers

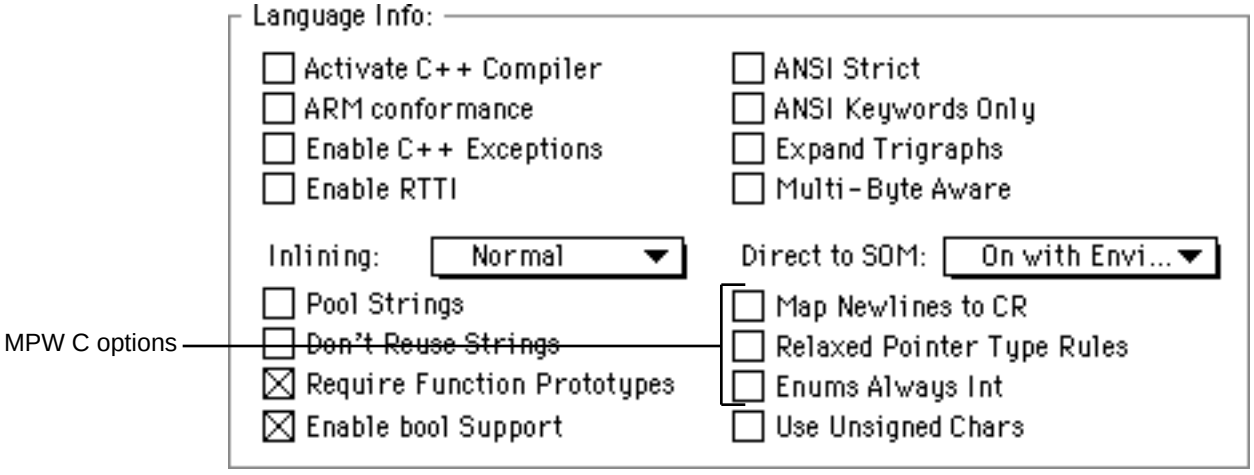
The MC68881 has eight registers, FP0 through FP7, which store 96-bit floating-point values (that is, `extended` or `long double`). If you turn on the **68881 Codegen** option, your assembly language routines can use registers FP0 through FP3 for temporary storage without restoring their values. If you use registers FP4 through FP7, you must preserve their contents.

The compiler allocates variables of type `long double` or `extended` to registers to optimize performance.

Calling MPW Functions

The CodeWarrior IDE lets you include an MPW C library in your CodeWarrior project and call most of its functions. You can set various options in the Processor and Language preference panel to make your project compatible with MPW C code. The Language preference panel options are shown in Figure 2.7.

Figure 2.7 MPW C Options in the C/C++ Languages Settings Panel



NOTE: The Win32/x86 compiler also honors the **Map Newlines to CR**, **Relaxed Pointer Type Rules**, and **Enums Always Int** options. However, it does not use the **MPW C Calling Convention** option.

Note that even if you turn on the **MPW C Calling Convention** option, MPW and Metrowerks aren't completely compatible in certain situations. For more information, see "Declaring MPW C functions (Macintosh Only)" on page 75.

This section contains the following:

WARNING! Do not turn off the **MPW C Calling Convention** or **Map Newlines to CR** options in Magic Cap code. Although the Magic Cap compiler lets you change the settings of these options, your code will not run correctly if they're off. They are on by default.

Adding an MPW library to a CodeWarrior project

To call a function from an MPW library, do the following.

1. **Add the library to your project with the Add Files command in the Project menu.**
2. **If you're using a 68K library, turn on the MPW C Calling Convention option.**

You can either turn on the **MPW C Calling Convention** option in the Processor preference panel, or you can use the pragma `mpwc`. If you use the **MPW C Calling Convention** option, all functions in your project use MPW C calling conventions. If you use the pragma

`mpwc`, only those functions declared with that pragma use MPW C calling convention.

To use the pragma, turn on the pragma `mpwc` in the header file that declares the MPW C functions, declare the functions, and turn off the pragma `mpwc`. For example:

```
#pragma mpwc on

int func1(double a, int b);
int func2(int a, double b);

#pragma mpwc reset
```

For more information, see "Declaring MPW C functions (Macintosh Only)" on page 75.

3. **If you're creating a 68K project, turn on the 4-Byte Int option in the Processor preference panel.**

   MPW C does not support 2-byte ints. For more information, see "Number Formats" on page 30.

4. **If you use the ANSI library to perform input or output, turn on the Map Newlines to CR option in the Language preference panel.**

   MPW and Metrowerks C and C++ handle the newline character ('\n') differently. For more information, see "Using MPW C newlines" on page 76.

5. **If your code relies on MPW C's relaxed type checking, turn on the Relaxed Pointer Type Rules option in the C/C++ Language settings panel .**

   Metrowerks C and C++ uses stricter rules than MPW when deciding whether certain pointer types are equivalent. For more information, see "Relaxing pointer checking" on page 56.

Declaring MPW C functions (Macintosh Only)

When you turn on the **MPW C Calling Convention** option, the compiler does the following to be compatible with MPW C's 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 off).
- Returns any 1-byte, 2-byte, or 4-byte structure in `D0`.
- If the **68881 Codegen** option is on, returns any floating-point value in `FP0`.

NOTE: The **MPW C Calling Conventions** option is available only with the 68K compilers. The PowerPC compilers don't need it, since all PowerPC compilers use the same calling conventions.

Note that even if you turn on the **MPW C Calling Convention** option, MPW and Metrowerks aren't completely compatible in these situations:

- Metrowerks C++ and MPW C++ classes are generally not compatible. Unless you follow the directions in "Declaring MPW-Compatible Classes" on page 104, you cannot use a Metrowerks C++ library in MPW or an MPW C++ library in a CodeWarrior project. If you need to use an MPW C library with Metrowerks C++ code, don't turn on the **MPW C Calling Conventions** option. Instead use the pragma `mpwc` as needed for non-member functions.
- To use MPW C functions that return a floating-point value, you must turn on the **68881 Codegen** option. If that option is off, Metrowerks C returns a long double value in a temporary variable, while MPW C returns it in a register.

This option corresponds to the pragma `mpwc`, described at "mpwc (68k Macintosh only)" on page 200. To check whether this pragma is on, `__option(mpwc)`, described at "mpwc (68K only)" on page 233. By default, this option is off.

Using MPW C newlines

If you turn on the **Map Newlines to CR** option in the Language preference panel, the compiler uses the MPW conventions for the `'\n'` and `'\r'` characters. If this option is off, the compiler uses the Metrowerks C and C++ conventions for these characters.

In most compilers, including Metrowerks C and C++, `'\r'` is translated to the value `0x0D`, the standard value for carriage return, and `'\n'` is translated to the value `0x0A`, the standard value for linefeed. However, in MPW C, `'\r'` is translated to `0x0A` and `'\n'` is translated to `0x0D`. When you turn on the **Map Newlines to CR** option, Metrowerks C conforms to MPW C conventions for these characters.

If you want to turn this option on, be sure you use the ANSI C and C++ libraries that were compiled with this option on. The 68K versions of these libraries are marked with an N; for example, `ANSI`

`(N/2i) C.68K.Lib`. The PowerPC versions of these libraries are marked with NL; for example, `ANSI (NL) C.PPC.Lib`.

If you turn this option on and use the standard ANSI C and C++ libraries, you won't be able to read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` brings you to the beginning of the current line instead of inserting a new line.

This option corresponds to the pragma `mpwc_newline`, described at "mpwc_newline" on page 201. To check whether this option is on, use `__option (mpwc_newline)`, described at "mpwc_newline" on page 233. By default, this option is off.

Calling Macintosh Toolbox Functions (Macintosh Only)

Metrowerks C and C++ let you use any routine described in *Inside Macintosh*. Simply call a routine exactly as it appears. Use these rules to convert the Pascal calling conventions to C:

- To pass a structure that is smaller than or equal to 4 bytes (such as a `Point`, `Cell`, or `Rect`), pass the actual structure.
- To pass a structure larger than 4 bytes, pass a pointer to the structure.
- To pass a `VAR` argument, pass a pointer that argument.
- To pass a string, pass a Pascal string.
- To pass any ResType or OSType, such as `'MENU'` or `'TEXT'`, pass a character literal.

The rest of this section describes creating Pascal strings, using Pascal variant records in the Macintosh Toolbox, and writing Pascal functions for the PowerPC:

- "Passing string arguments" on page 78
- "Using the pascal keyword in PowerPC code" on page 79

Passing string arguments

Metrowerks C and C++ have two kinds of string parameters: C strings and Pascal strings. Most C functions, such as the ANSI libraries, use C strings, arrays of characters whose last element is the null byte (`\0`). Most Pascal routines, such as the Macintosh Toolbox, use Pascal strings, arrays of characters whose initial element is the number of characters in the string.

To create a Pascal string literal, use `\p` at the beginning of the string. For example, this statement sets the title of a window:

```
SetWTitle (myWinPtr, "\pMy window");
```

To declare a variable or argument that is a Pascal string, use one of these types: `Str255`, `Str63`, `Str32`, `Str31`, `Str27`, `Str15`. The number in the type's name specifies the number of characters that the string may contain. For example, this statement declares a Pascal string with 255 characters:

```
Str255 winTitle;
```

Since both string formats have an extra byte of information (either a count at the beginning or a null byte at the end), the compiler can transform a string in place from Pascal to C and vice versa. The routines `c2pstr()` and `p2cstr()`, declared in the header file `Strings.h`, perform these conversions. They are declared like this:

```
char *p2cstr(StringPtr aStr);
StringPtr c2pstr(char *aStr);
```

The following example creates a window title that contains the name of the current user. It gets the name of the user from a Pascal routine, creates the window title with a C routine, and sets the window title with a Pascal routine:

```
char* winTitle[256];
Str32 userName;

err = GetDefaultUser(&ref, &userName);
sprintf(winTitle, "%s's window",
        p2cstr(userName));
SetWTitle(myWinPtr, c2pstr(winTitle));
```

Generally, Macintosh Toolbox routines expect a string argument to be a Pascal string. However, the universal headers sometimes declare two versions of a function: one that uses C strings and one that uses Pascal strings. When you come across a function like this, follow these rules:

- If a Macintosh Toolbox routine name is all lower-case, use C strings.
- If a Macintosh Toolbox routine name contains a mixture of upper-case and lower-case letters, use Pascal strings.

For example, `SetWTitle()` expects a Pascal string:

```
SetWTitle (myWinPtr, "\pMy window");
```

And `setwtitle()` expects a C string:

```
setwtitle (myWinPtr, "My window");
```

Using the pascal keyword in PowerPC code

Since the PowerPC handles `pascal` functions differently from 68K, you must be careful when you're writing a filter or call-back function that works with a Macintosh Toolbox function. If your function takes an argument which is a structure larger than 4 bytes, you *must* declare that argument as a pointer to the structure. For example:

```
pascal OSErr MyOapp( AppleEvent aevt,
      AppleEvent reply, long refCon );
  // WRONG: On PPC, aevt and reply will
  // point to garbage. Code may work on 68K.

pascal OSErr MyOapp( AppleEvent *aevt,
      AppleEvent *reply, long refCon );
  // OK: Code will work on both PPC and 68K.
```

You were always encouraged to declare a large structure argument as a pointer to the structure. But since the 68K would pass the structure on the stack anyway, you could get away with declaring a large structure argument as the structure itself. However, the PowerPC is much stricter and never passes a structure larger than 4 bytes on the stack.

Intrinsic PowerPC Functions (Macintosh Only)

Metrowerks C/C++ for PowerPC provides intrinsic functions to generate inline PowerPC instructions. These intrinsic functions are faster than other functions, since the compiler translates them into inline assembly instructions instead of function calls.

NOTE: These intrinsic functions are not part of the ANSI C or C++ standards. They are available only with the Metrowerks C/C++ for PowerPC compiler. They are not available with the Metrowerks C/C++ for 68K compiler.

This section contains the following:

Low-level processor synchronization

These functions perform low-level processor synchronization.

```
void __eieio(void)
/* Enforce In-Order Execution of I/O         */

void __sync(void)
/* Synchronize                               */

void __isync(void)
/* Instruction Synchronize                   */
```

For more information on these functions, see the instructions `eieio`, `sync`, and `isync` in *PowerPC Microprocessor Family: The Programming Environments* by Motorola.

Floating-point functions

These functions generate inline instructions that take the absolute value of a number.

```
int __abs(int);
/* Absolute value of an integer.             */

float __fabs(float);
/* Absolute value of a float.                */

float __fnabs(float);
/* Negative of the absolute value of a float.*/

long __labs(long);
/* Absolute value of a long int.             */
```

Byte-reversing functions

These functions generate inline instructions than can dramatically speed up certain code sequences, especially byte-reversal operations

```
int __cntlzw(int);
/* Count leading zeros in a integer.         */

int __lhbrx(void *, int);
/* Load half word byte – reverse indexed.    */

int __lwbrx(void *, int);
/* Load word byte – reverse indexed.         */

void __sthbrx(unsigned short, void *, int);
/* Store half word byte – reverse indexed.   */

void __stwbrx(unsigned int, void *, int);
/* Store word byte – reverse indexed.        */
```

Setting the floating-point environment

This function lets you change the PowerPC processor's Floating Point Status and Control Register (FPSCR). It sets the FPSCR to its argument and returns the original value of the FPSCR.

```
float __setflm(float);
```

This example shows how to set and restore the FPSCR:

```
double old_fpscr;
oldfpscr = __setflm(0.0);
  /* Clear all flag/exception/mode bits and
   * save the original settings.            */

/* . . .
 * Peform some floating point operations
 */

__setflm(old_fpscr);
  /* Restore the FPSCR.                     */
```

Floating-point instructions for the 603 and 604

These floating-point instructions, which are available only on the PowerPC 603 and 604, can speed up certain types of graphics code.

WARNING! On a Mac OS computer with a PowerPC 601, they will raise an illegal instruction exception and may crash your program.

```
float __fres(float);
  /* Floating Reciprocal Estimate Single      */

double __fsqrte(double);
  /* Floating Reciprocal Square Root Estimate */

double __fsel(double, double, double)
  /* Floating Select                          */
```

Rotating the contents of a variable

These functions rotate the contents of a variable to the left.

```
int __rlwinm(int, int, int, int);
  /* Rotate Left Word Immediate
     then AND with Mask         */

int __rlwnm(int, int, int, int);
  /* Rotate Left Word then AND with Mask */

int __rlwimi(int, int, int, int, int);
  /* Rotate Left Word Immediate
     then Mask Insert           */
```

Please note that the first argument to `__rlwimi` is overwritten.

3

C++ Language Notes

This chapter describes how Metrowerks C++ handles the parts of the C++ language that are unique to C++ and not shared by C.

Overview of C++ Language Notes

This chapter describes how Metrowerks C++ handles the parts of the C++ language that are unique to C++ and not shared by C. For more information on the parts of the language that C and C++ share, see “Overview of C and C++ Language Notes” on page 19.

In the margins of this chapter are references to ARM, which is *The Annotated C++ Reference Manual* (Addison-Wesley) by Ellis and Stroustrap. These references show you where to look for more information on the topics discussed in the near-by section.

This chapter contains the following sections:

- “Unsupported Extensions” on page 86 describes some common extensions to the C++ standard that Metrowerks C++ does not currently support.
- “Metrowerks Implementation of C++” on page 86 describes how Metrowerks C++ implements certain sections of the C++ standard.
- “Setting C++ Options” on page 92 describes how to change Metrowerks C++’s behavior by setting options in the C/C++ Language settings panel.
- “Using Run-Time Type Information (RTTI)” on page 96 describes the `dynamic_cast` and `typeid` operators.
- “Using Templates” on page 99 describes the best way set up the files that define and declare your templates. It also documents an addition to the C++ standard which lets you explicitly instantiate templates.

- "Using Exceptions" on page 103 describes how to use the `try` and `catch` statements to perform exception handling.
- "Declaring MPW-Compatible Classes" on page 104 describes how to create classes you can use in libraries for either MPW C++ or Metrowerks C++.
- "Creating Direct-to-SOM Code" on page 105 describes how to write SOM code with Metrowerks C++.

Unsupported Extensions

The C++ compiler does not currently support these common extension to *The Annotated C++ Reference Manual* (Addison-Wesley) by Ellis and Stroustrap:

- Overloading methods `operator new[]` and `operator delete[]`, which let you allocate and deallocate the memory for a whole array of objects at once. Instead, overload `operator new()` and `operator delete()`, which are the functions that `operator new[]` and `operator delete[]` call. (ARM, §5.3.3, §5.3.4)
- Name spaces
- The `mutable` keyword

Metrowerks Implementation of C++

This section describes how Metrowerks C++ implements certain parts of the C++ standard, as described in *The Annotated C++ Reference Manual* (Addison-Wesley) by Ellis and Stroustrap. It contains the following:

- "Checking for resources to initialize static data" on page 90
- "Calling an inherited member function" on page 91

Which keywords to put first

(ARM §7.1.2, §11.4) If you use either the `virtual` or the `friend` keyword in a declaration, it must be the first word in the declaration. For example:

Listing 3.1 Using the virtual or friend keywords

```
class foo {
  virtual int f0();  // OK
  int virtual f1();  // ERROR

  friend int f2();   // OK
  int friend f3();   // ERROR
}
```

Additional keywords

(ARM §2.4, ANSI §2.8) In addition to reserving the symbols in §2.3 of the ARM as keywords, Metrowerks C++ reserves these symbols from §2.8 of the ANSI Draft C++ Standard as keywords:

`bool`	`const_cast`	`dynamic_cast`
`explicit`	`false`	`mutable`
`namespace`	`reinterpret_char`	`static_cast`
`true`	`typeid`	`using`

Metrowerks C++ does not implement the symbol `wchar_t` from §2.8 of the ANSI Draft C++ Standard.

Conversions in the conditional operator

(ARM §5.16) The compiler does not apply reference conversions to the second and third expressions of the conditional operator. In

other words, unless the second and third expressions are numeric types, they must be the same type.

Listing 3.2 A conversion in a conditional operator

```
class base { };
class derived : public base { };

static void foo(derived i)
{
  base      &a = i;
  derived   &b = i, c;

  c = (sizeof(0)?a:b);
    // ERROR: b is not converted to (base &)

  c = (sizeof(0)?a:(base &)b)
    // OK
}
```

Default arguments in member functions

(ARM, §8.2.6) 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 must be declared. For example:

Listing 3.3 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
  enum B { BB };   //        yet
};
```

Local class declarations with inline functions

(ARM, §9.8) If you're declaring a class within a function, the class's inline functions cannot access the outer function's local types or variables. In other words, the compiler inserts the class's inline functions on global scope level. For example:

Listing 3.4 Using local class declarations with inline functions

```
int x;

void foo()
{
  static int s;

  class local {
    int f1() { return s; }
            // ERROR: cannot access 's'

    int f2() { return local::f1(); }
            // ERROR: cannot access local

    int f3() { return x; }
            // OK
  };
}
```

Copying and constructing class objects

(ARM, §12.1, §12.8) The compiler does not generate a copy constructor or a default `operator=` for a simple class. A simple class is a class that:

- Is a base class or is derived only from simple classes
- Has no class members or has only simple class members
- Has no virtual member functions

- Has no virtual base classes
- Has no constructor or destructor

Listing 3.5 Constructors

```
class Simple { int f; };

void simpleFunc (Simple s1)
{
  Simple s2=Simple(s1);
            // ERROR: An explicit copy constructor
            //        call. The compiler generates
            //        no default copy constructor.

  Simple s3=s1;
            // OK: The compiler performs a
}            //     bitwise copy
```

The compiler does not guarantee that generated assignment or copy constructors will assign or initialize objects representing virtual base classes only once.

Checking for resources to initialize static data

Sometimes you create static C++ objects that require certain resources, such as a floating-point unit (FPU). You can check for these resources by creating a function called `__PreInit__()` which the compiler calls before it initializes static data. You cannot check for these resources in your `main()` routine, since the compiler initializes static data before it calls `main()`.

You must declare the `__PreInit__()` function like this:

```
extern "C" void __PreInit__(void);
```

NOTE: The PPC compiler does not support this function.

This stub checks for a floating-point unit: (Note that you must define the functions `HasFPU()` and `DisplayNoFPU()` yourself.)

Listing 3.6 Checking for an FPU before initializing static data

```
#include <Types.h>
#include <stdlib.h>

extern "C" void __PreInit__(void);

void __PreInit__(void)
{
  if(!HasFPU())   {
    DisplayNoFPU();  // Display "No FPU" Alert
    abort();         // Abort program exection
  }
}
```

Calling an inherited member function

(ARM, §10.2) Metrowerks C++ lets you incrementally build upon a class's behavior with the `inherited` keyword. Frequently when you override a function, you just want to add some behavior to the overridden function. Metrowerks C++ lets you call the overridden function with the `inherited` keyword and then perform the additional behavior. The syntax is the following:

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

The statement calls the *func-name* that the class's base class would call. If class has more than one base class and the compiler can't 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:

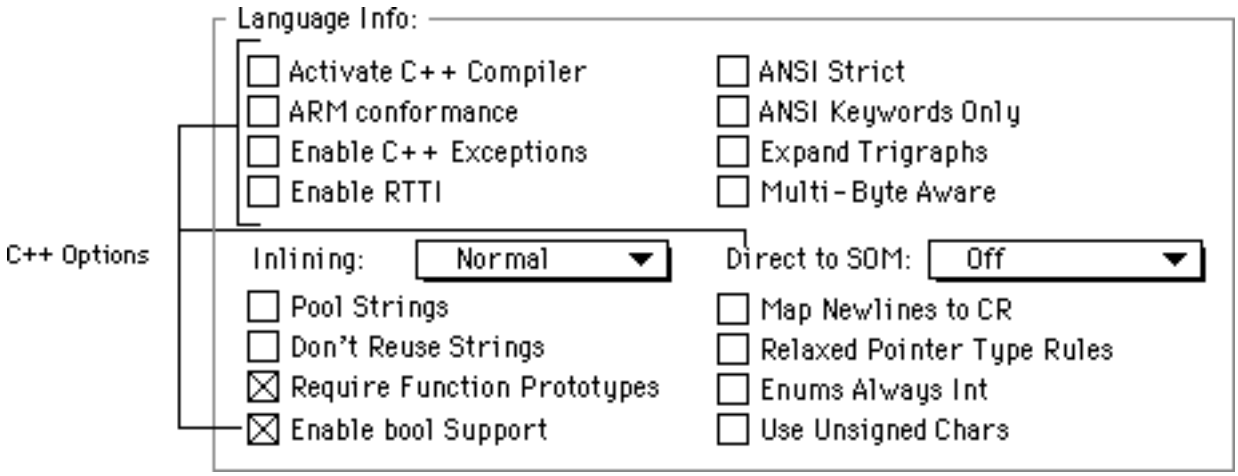
Listing 3.7 Using the inherited keyword to call an inherited member function

```
class O { virtual void draw(Point); }
class Q : O { void draw(Point); }

void O::draw (Point p)
{
  Rect r = { p.x-5, p.y-5, p.x+5, p.y+5 };
  FrameOval(r);        // Draw an O.
}


void Q::draw (Point p)
{
  inherited::draw(p);   // Perform behavior of
                        //   base class
  MoveTo(p.x, p.y);     // Perform added behavior
  Line(5, 5);
}
```

Setting C++ Options

This section describes how to change the behavior of Metrowerks C++ by setting some options in the Language preference panel. Figure 3.1 shows where the C++ options are. For information on the rest of the options in the C/C++ Language settings panel, see "Overview of C and C++ Language Notes" on page 19.

Figure 3.1 Setting C++ Options in the C/C++ Languages Settings Panel



This section contains the following:

For more information on Direct to SOM, see "Creating Direct-to-SOM Code" on page 105.

Using the C++ compiler always

If you turn on the **Activate C++ Compiler** option, the compiler compiles all the C source files in your project as C++ code. If you turn this option off, the CodeWarrior IDE looks at a file name's suffix to determine whether to use the C or C++ compiler. These are the suffixes it looks for:

- If the suffix is `.cp`, `.cpp`, or `.c++`, the CodeWarrior IDE uses C++
- If the suffix is `.c`, the CodeWarrior IDE uses C.

This option corresponds to the pragma `cplusplus`, described on "cplusplus" on page 179. To check whether this option is on, use

`__option (cplusplus)`, described on "cplusplus" on page 230. By default, this option is off.

Enforcing strict ARM conformance

When the **ARM Conformance** option is on, Metrowerks C++ generates an error when it encounters certain ANSI C++ features that conflict with the C++ specification in *The Annotated C++ Reference Manual*. Use this option only if you must make sure that your code strictly follows the specification in *The Annotated C++ Reference Manual*.

Turning on this option prevents you from doing the following

- Using protected base classes (ARM, §11.2). For example:

```
class X {};
class Y : protected X {};
  // OK in Metrowerks C++. Error in ARM.
```

- Changing the syntax of the conditional operator to let you use assignment expressions without parentheses in the second and third expressions (K&R, §A7.16). For example:

```
i ? x=y : y=z
  // OK in Metrowerks C++. Error in ARM
i ? (x=y):(y=z)
  // OK in ARM and Metrowerks C++
```

- Declaring variables in the conditions of `if`, `while` and `switch` statements (K&R, §A9.4, §A9.5). For example:

```
while (int i=x+y) { . . . }
  // OK in Metrowerks C++. Error in ARM.
```

Turning on this option *allows* you to do the following:

- Using variables declared in the condition of an `for` statement after the `for` statement (K&R, §9.5). For example:

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

This option corresponds to the pragma `ARM_conform`, described on "ARM_conform" on page 174. To check whether this option is on,

use `__option (ARM_conform)`, described on "ARM_conform" on page 230. By default, this option is off.

Adding C++ extensions

If you turn on the pragma `cpp_extensions`, the compiler lets you use these extensions to the ANSI C++ standard:

- Anonymous structs (ARM, §9). For example:

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

- Unqualified pointer to a member function (ARM, §8.1c). For example:

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

  void (Foo::*ptmf2)() = f;
    // OK, if cpp_exptensions is on.
}
```

This pragma does not correspond to any option in the preference panel. To check whether this option is on, use the `__option (cpp_extensions)`, described on "Options Checking" on page 229. By default, this option is off.

Allowing exception handling

Turn on the **Enable C++ Exceptions** option if you use PowerPlant or the ANSI-standard `try` and `catch` statements. Otherwise, turn off this option to generate smaller and faster code.

TIP: For more information on Metrowerks implements ANSI C++'s exception handling mechanism, see "Using Exceptions" on page 103.

This option corresponds to the pragma `exceptions`, described on "exceptions (C++ only)" on page 185. To check whether this option is on, use `__option (exceptions)`, described on "exceptions" on page 231. By default, this option is off.

Using the bool type

Turn on the **Enable bool Support** option if you want to use the standard C++ `bool` type to represent `true` and `false`. Turn this option off if recognizing `bool`, `true`, or `false` as keywords would cause problems in your program.

This option corresponds to the pragma `bool`, described on "bool (C++ only)" on page 176. To check whether this option is on, use `__option (bool)`, described on "bool" on page 230. By default, this option is off.

Using Run-Time Type Information (RTTI)

Metrowerks C++ supports Run-Time type Information (or RTTI), including the `dynamic_cast` and `typeid` operators. To use these operators, turn on the **Enable RTTI** option in the C/C++ Language preference panel.

The rest of this section describes the two parts of RTTI:

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 may crash your program if the conversion is not possible

This is the syntax for `dynamic_cast` operator:

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

The *Type* must be either `void` or a class with at least one virtual function member. If the object that *expr* points to (`*`*expr*) is of type *Type* or is 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` would work with each:

```
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 refer-

ences, `dynamic_cast` throws an exception of type `bad_cast` if it cannot perform the conversion.

NOTE: The `bad_cast` type is defined in the header file exception. Whenever you use `dynamic_cast` with a reference, you must #include exception.

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

```
#include <exception>
// . . .
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 `type_info` object that you can compare with the == and `!=` operators. For example, take these classes from above:

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

```
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.
if (typeid(*arnold) == typeid(*louganis)) //...
  // arnold and louganis are both Athletes.
if (typeid(*lois) != typeid(*arnold)) // ...
  // lois and arnold are not the same type.
```

You can access the name of a type with the `name()` member function in the `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
```

Using Templates

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

This section contains the following:

Declaring and defining templates

In a header file, declare your class functions and function templates, as shown in Listing 3.8.

Listing 3.8 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, and define the function templates and the member functions of the class templates, as shown in Listing 3.9. This is a template definition file. You'll include this file in any file that uses your templates. You do not need to add the template definition file to your project.

Listing 3.9 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);
}
```

NOTE: Although the template definition file is a source file and ends in `.cp`, it is the file you will include in any other source file that uses your templates. If you include the template declaration file, which ends in `.h`, the compiler will generate an error saying that the function or class is undefined.

Instantiating templates

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 templates arguments. Specifying these values is called instantiating the template.

Metrowerks 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'll need in one place:

- If you use automatic instantiation, the compiler may take longer to compile your program since it has to determine on its own which instantiations you'll need. Also, the object code for the template instantiations will be scattered throughout your program.
- If you use explicit instantiation, the compiler compiles your program quicker. Since the instantiations can be in one file, with no other code, you can choose to put them all in one segment or even in a separate library.

NOTE: Explicit instantiation is not in the ARM but is part of the ANSI C++ draft standard.

To instantiate templates automatically, include the template definition file in all the source files that use the templates, and just use the templates as you would any other type or function. The compiler automatically generates code for a template instantiation whenever it sees a new one. Listing 3.10 shows how to automatically instantiate the templates in Listing 3.8 and Listing 3.9.

Listing 3.10 myprog.cp: A Source File that Uses Templates

```
#include <iostreams.h>
#include "templ.cp"
    // This statement includes both the template
    // declarations and the template defintions.

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.
}
```

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

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

The syntax for a function template instantiation is

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

Listing 3.11 shows how to explicitly instantiate the templates in Listing 3.8 and Listing 3.9.

Listing 3.11 myinst.cp: Explicitly Instantiating Templates

```
#include "templ.cp"

template class Templ<long>;
  // class instantiation

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

When you're explicitly instantiating 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 3.11 you can instantiate `Max<long>()` like this:

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

Using Exceptions

If you turn on the **Enable C++ Exceptions** options in the C/C++ Languages preference panel, you can use the `try` and `catch` statements to perform exception handling. If your program doesn't use exception handling, turn this option to make your program smaller.

You can throw exceptions across any code that's compiled by the CodeWarrior 8 (or later) Metrowerks C/C++ compiler with the **Enable C++ Exceptions** option turned on. You cannot throw exceptions across the following:

- Macintosh Toolbox function calls
- Libraries compiled with the **Enable C++ Exceptions** option turned off
- Libraries compiled with versions of the Metrowerks C/C++ compiler earlier than CodeWarrior 8
- Libraries compiled with Metrowerks Pascal or other compilers.

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

If you throw an exception when you're 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.

Declaring MPW-Compatible Classes

Metrowerks C++ lets you declare classes that save you some overhead and that are automatically created on the application's heap. These classes are also the only type of Metrowerks C++ classes that are compatible with MPW C++ code. Use them only when you need to save as much space as possible or need to create a library you can use with MPW C++.

These are the two types of objects:

- SingleObject objects are created on the stack.
- HandleObject objects are created in the application's heap.

TIP: For more information on writing MPW-compatible C code, see "Calling MPW Functions" on page 72.

Since these classes do not let you use multiple-inheritance or run-time type information (RTTI), they can save you some overhead. The compiler stores information about an object's virtual functions in a data structure called a *virtual table*. The virtual table for a single-inheritance object can be much simpler and smaller than the one for a multiple-inheritance object.

HandleObject has all the features as SingleObject, with one additional feature: Any object descended from it is automatically stored on the application's heap, and you reference the object with a handle. You treat these handles as pointers, since the compiler automat-

ically changes the pointer references to handle references for you. For example:

```
class myClass : HandleObject {
    int a;
    // . . .
}

MyClass *myObj = new MyClass
myObj->a = 0;
// The compiler automatically converts these
// pointer references to handle references.
```

These restrictions apply to objects descended from HandleObject:

- You cannot use multiple inheritance or run-time type information.
- You must create a new HandleObject object with the `new` operator.
- You cannot create a HandleObject local variable, global variable, array, class member , or function parameter. However, HandleObject *pointers* can be any of the above.
- You cannot cast a HandleObject pointer to another type, other than a pointer to another HandleObject object. You cannot cast any other type of pointer to a HandleObject pointer.
- When you dereference a HandleObject pointer , you can use it only to refer to a class member. For example:

```
myObj->a = 0;              // OK
*myObj.a = 0;              // OK
func( *myObj );           // ERROR
```

- Avoid taking the address of a member of a HandleObject object (such as `&myObj->a`). Since the object is in the heap, it may move unexpectedly and the address will point to garbage.

Creating Direct-to-SOM Code

Metrowerks C/C++ lets you create SOM code directly in the CodeWarrior IDE. SOM is an integral part of OpenDoc.

There are two ways to create SOM code. You can turn select **On** or **On with Environment Checks** from the **Direct to SOM** menu in the C/C++ Language preference panel, or use the `direct_to_som` pragma before you import any SOM header files, like this:

```
#pragma direct_to_som on
```

If you select **On with Environment Checks** from the **Direct to SOM** menu, the compiler performs some automatic error checking, as described in "Automatic SOM error checking" on page 109.

Note that when you turn on the **Direct to SOM** option, you should turn on the **Enums Always Int** option in the C/C++ Language preference panel, described in "Enumerated constants of any size" on page 51.

Also, when you define a SOM class, Metrowerks C/C++ uses PowerPC alignment for that class. In other words, the compiler acts as though you enclosed the class definition with `#pragma options align=powerpc` and `#pragma options align=reset`. For more information on structure alignment, see *Targeting Mac OS*.

The rest of this section describes the restrictions SOM code must abide by, some useful SOM header files, and pragmas for SOM classes:

SOM class restrictions

Since you can develop SOM code in different languages and then use that code under different operating systems, you must work with several restrictions when developing SOM code.

These restrictions apply only to classes that are descended from SOMObject. You can use SOMObjects and other classes together in a project.

When you create a SOM class and define its members, keep these restrictions in mind:

- The base class must be SOMObject or a descendant of SOMObect. If you use multiple inheritance, all parent classes must be descendants of SOMObject. (You cannot mix SOM classes with other classes in the base list for any class.)
- You must declare the class with the `class` keyword. A class declared as `struct` or `union` cannot be a SOM class.
- All the class inheritance must be `virtual`.
- All the class's data members must be `private`.
- The only member functions you can overload are inline member functions that are not virtual. They are not considered to be SOM methods.
- The only operations you can overload are `new` and `delete`.
- The class must contain at least one member function that's not inline. MacSOM uses the first such class to determine whether the class is implemented in a particular compilation unit.
- The class cannot contain the following:
  - nested class definitions
  - static data or function members.
  - constructors (ctors) with parameters.
  - copy constructors
- In a member function, you cannot do the following:
  - use `long double` parameters or return type
  - use a variable length argument list

When you use a SOM class in your code, remember that you *cannot* do the following:

- Create global SOM objects.
- Use `sizeof()` with SOM objects or classes.
- Create class templates that expand to SOM objects.
- Create arrays of SOM objects.

- Use the placement and array forms of new (such as `new(address) T` or `new T[n]`) or the array form of delete (such as `delete [] p`).
- Declare SOM classes as members of other classes. (You can declare pointers to SOM class objects as members.)
- Take the address of a member of a SOM class. For example, `&foo::bar` is not allowed if `foo` is a SOM class.
- Pass aggregate parameters by value to a SOM member function.
- Use SOM objects as function parameters. (You can use a pointer to a SOM object as a parameter.)
- Perform an assignment with SOM classes
- Return a SOM object as a function's value

Also when you invoke a method with explicit scope (such as `obj->B::func()`), the specified class (`B`) must be the same class as the object (`obj`) or a direct parent of the object's class.

For example, if class A is the parent of class B which is the parent of class C, then

```
C* obj = new C;

obj->C::func();  // OK: C is obj's class
obj->B::func();  // OK: B is a direct parent
                 //     of obj's class
obj->A::func();  // ERROR: A is NOT a direct
                 //        parent of obj's class
```

Using SOM headers

CodeWarrior includes several different header files for use in SOM code. These are the most important and probably the only ones you'll need to use yourself:

Table 3.1 SOM Headers

This header	Contains this…
`somobj.hh`	SOMObject, a SOM base class. If your file sub-classes from SOMObject, include this header. If you're converting a file from IDL to Metrowerks C++, you can use this header as a replacement for `somobj.idl` and `somobj.xh`.
`somcls.hh`	SOMClass, the SOM base meta-class. If your file sub-classes from SOMClass, include this header. If you're converting a file from IDL to Metrowerks C++, you can use this header as a replacement for `somcls.idl` and `somcls.xh`.
`som.xh`	The procedural interface to SOMObjects for Mac OS. It's not needed for basic SOM programming.
`somobj.xh`	Same as `somobj.hh`. Use `somobj.hh` instead.
`somcls.xh`	Same as `somcls.hh`. Use `somcls.hh` instead.

Automatic SOM error checking

If you choose **On with Environment Checks** from the **Direct to SOM** menu, the compiler performs some automatic error checking, in addition to creating SOM code. 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 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 method's result.

And, the compiler transforms this new allocation:

```
new SOMclass;
```

into something that is equivalent to 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.

You must define `__som_check_ev()` and `__som_check_new()` to do something like this:

Listing 3.12 The __som_check_ev() and __som_check_new() functions

```
#include <somdts.h>
#pragma internal on

extern "C" void __som_check_ev(
      struct Environment * );
extern void __som_check_ev(
      struct Environment *envp )
{
  if(envp->_major)
  {
    // your error handling code here
  }
}
```

```
extern "C" void __som_check_new( SOMObject * );
extern void __som_check_new( SOMObject *SOMObj)
{
  if(somp==NULL)
  {
    // your error handling code here
  }
}
```

The PowerPC compiler uses an optimized error check that is smaller but slightly slower than the one given above. To use the error check show above in PowerPC code, use the pragma `SOMCall-Optimization`. It looks like this:

```
#pragma SOMCallOptimization on | off | reset
```

The default is on.

You can also turn on SOM error checking with with this pragma:

```
#pragma SOMCheckEnvironment on | off | reset
```

The default is `off`.

Using SOM pragmas

The following pragmas let you give information on a SOM class to the MacSOM software:

- `SOMReleaseOrder` declares the release order of a class's methods.
- `SOMClassVersion` declares the version number for a class.
- `SOMMetaClass` declares the metaclass for a class.
- `SOMCallStyle` declares the call style (IDL or OIDL) for a class.

All pragmas besides `SOMCheckEnvironment` must appear within the declaration of the class they apply to. These pragmas may appear more than once in a class declaration, but they must specify the same information each time.

Declaring the release order

A SOM class must specify the release order of its member functions. As a convenience for when you're first developing the class, Metrowerks C++ 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, since you'll need to modify its list in later versions of the class. The pragma looks like this:

```
#pragma SOMRelaseOrder(func1, func2,... funcN)
```

You must specify every SOM method that the class introduces. Do not specify inline member functions that are not virtual, since they're not considered to be SOM methods. Don't 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.

Declaring the class's version

SOM uses the class's version number to make sure the class is compatible with other software you're using. If you don't declare the version numbers, SOM assumes zeroes.

The `SOMClassVersion` pragma looks like this:

```
#pragma SOMClassVersion(class,majorVer,minorVer)
```

The version numbers must be positive or zero.

When you define the class, the program passes its version number to the SOM kernel in the class's 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.

Declaring the metaclass for a class

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. It looks like this:

```
#pragma SOMMetaClass (class, metaclass)
```

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.

Declaring the call style for a class

SOM supports two call styles:

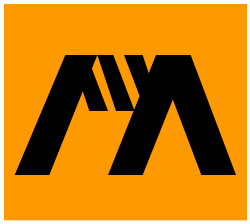
- OIDL, an older style that does not support DSOM
- IDL, a newer style that does support DSOM.

By default, Metrowerks C++ assumes that a class uses IDL. To use OIDL, use the `SOMCallStyle` pragma, which looks like this:

```
#pragma SOMCallStyle OIDL
```

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.

4



68K Assembler Notes

This chapter describes the 68K assembler that is part of the CodeWarrior package of compilers.

Overview of 68K Assembler Notes

Frequently you want to include a small amount of assembly code in a program. For example, you may want to make sure that a frequently-used function is written as efficiently as possible. Both the PowerPC and 68K compilers include built-in assemblers that let you do just that.

This chapter describes how to use the built-in 68K assembler with either the 68K Macintosh compiler or the Magic Cap compiler, including its syntax and special directives. It does not document all the instructions available in 68K assembler. For more information, see the *MC68000 Family Programmer's Reference Manual* from Motorola.

TIP: For more information on the built-in PowerPC assembler, see "Overview of PowerPC Assembler Notes" on page 125.

The topics in this chapter include:

Writing an Assembly Function for 68K

This section details how to write a function for the 68K assembler. The topics in this section include:

Defining a Function for 68K Assembly

To include assembly in your 68K project, declare a function with the `asm` qualifier, like this:

```
asm long f(void) { . . . } // OK: An assembly
                           //     function
```

Note that you cannot create an assembly statement block within a C function:

```
long f(void)
{
  asm { . . . }   // ERROR: Assembly statement
}                 // blocks are not supported.
```

The built-in assembler uses all the standard MC 680000 assembler instructions. It accepts some additional directives described in "Assembler directives" on page 122. It also accepts the following 68020 assembler instructions, after you use one of these directives: `machine 68020`, `machine 68030`, or `machine 68040`:

`bfchg`	`bfclr`	`bfexts`	`bfextu`
`bfffo`	`bfins`	`bfset`	`bftst`

`divsl`	`divs.l`	`divul`	`divu.l`
`muls.l`	`mulu.l`	`extb.l`	`rtd`

You cannot use MC68020, MC68030, or MC68040 addressing modes.

TIP: If you know the opcode for an assembly statement that's not supported, you can include it in your function with the `opword` directive, described at "opword" on page 124.

Keep these tips in mind as you write assembly functions:

- All statements must follow this syntax:

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

  Each instruction must end with a newline or a semicolon (;).
- Hex constants must be in C-style , not Pascal-style. For example:

```
move.l   0xABCDEF, d5    // OK
move.l   $ABCDEF, d5     // ERROR
```

- Assembler directives, instructions, and registers are not case-sensitive. For example these two statements are same:

```
move.l   b, DO           // OK
MOVE.L   b, d0           // ALSO OK
```

- A label must end in a colon and may contain the @ character. For example:

```
asm void foo(void)
{
x1:  dc.b     "Hello world!\n"  // OK
@x2: dc.w     5                 // OK
x3   dc.w     1,2,3,4    // ERROR: Needs a colon
}
```

- You cannot begin comments with a semicolon (;), but you can use C and C++ comments. For example:

```
add.l      d5,d5                ; ERROR
add.l      d5,d5                // OK
add.l      d5,d5                /* OK */
```

Listing 4.1 shows an example of an assembly function.

Listing 4.1 Creating an assembly function

```
long int b;
struct mystruct {
  long int a;
} ;

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

The rest of this section describes how to create local variables, access function parameters, refer to fields within a structure, and use the preprocessor with the assembler. A section at the end of the chapter describes some special assembler directives that the built-in assembler allows.

Using Global Variables in 68K Assembly

To refer to a global variable, just use its name, as shown below:

```
int x;
asm void f(void)
{
  move.w      x,d0    // Moving x into d0
  // . . .
}
```

Using Local Variables and Arguments in 68K Assembly

The built-in assembler gives you two ways to refer to local variables and function arguments: you can do the work on your own or let the built-in assembler do the work for you. To do it on your own, you must explicitly save and restore processor registers and local variables when entering and leaving your assembly function. You cannot refer to the variables by name. You can refer to function arguments off the stack pointer. For example, this function moves its argument into `d0`:

```
asm void foo(short n)
{
  move.w      4(sp),d0 //  n
  // . . .
}
```

To let the built-in assembler do it for you, use the directives `fralloc` and `frfree`. Just declare your variables as you would in a normal C function. Then use the `fralloc` directive. It makes space on the stack for the local stack variables and reserves registers for the local register variables (with the statement `link #`*x*`,a6`). In your assembly, you can refer to the local variables and variable arguments by name. Finally, use the `frfree` directive to free the stack storage and restore the reserved registers.

Listing 4.2 is an example of using local variables and function arguments.

Listing 4.2 Using the fralloc directive

```
static asm short f(short n)
{
  register short a; // Declaring a as a register
  short b;          // variable and b as a stack
                    // variable. Note that you need
                    // semicolons at the ends of
                    // these statements.

  fralloc +     // Allocate space on stack
                //  and reserve registers.
  move.w  n,a   // Using an argument and local var.
  add.w   a,a
  move.w  a,D0

  frfree        // Free the space that
                //  fralloc allocated
  rts
}
```

Using Structures in 68K Assembly

You can refer to a field in a structure with the `struct` construct, as shown below:

`struct(`*structTypeName*`.`*fieldName*`)` *structAddress*

This instruction moves into `D0` the `refCon` field in the `WindowRecord` that `A0` points to:

```
move.l  struct(WindowRecord.refCon) (A0), D0
```

Using the Preprocessor in 68K Assembly

You can use all preprocessor features, such as comments and macros, in the assembler. Just keep these points in mind when writing a macro definition:

- End each assembly statement with a semicolon (`;`), since the preprocessor ignores newlines. For example:

```
#define MODULO(x,y,result)\
          move.w   x,D0; \
          ext.l    D0; \
          divs.w   y,D0; \
          swap     D0; \
          move.w   D0,result
```

- Use `%` instead of #, since the preprocessor uses # as an operator to concatenate token. For example:

```
#define ClearD0  moveq %0,D0
```

Returning From a Function in 68K Assembly

Every assembly function should end in an `rts` or a `preturn` statement. If you forget to add one, the compiler does not add one for you, and does not raise an error. Use the `rts` statement for ordinary C functions. Use the `preturn` statement for Pascal functions, since it performs the clean up that Pascal functions need. For example:

```
asm void f(void)
{
    add.l      d4, d5
}                       // No RTS statement

asm void g(void)
{
    add.l      d4, d5
    rts                 // OK
}
```

```
asm void pascal h(void)
{
    add.l      d4, d5
    preturn             // OK
}
```

Assembler directives

This section describes some special assembler directives that the Metrowerks built-in assembler accepts. The directives are listed alphabetically.

dc

`dc[.(b|w|l)]` *constexpr* `(,`*constexpr*`)*`

Defines a block of constant expressions, *constexpr*, as initialized bytes, words, or long words. If there is no qualifier, `.w` is assumed. For `dc.b` you can specify any string constant (C or Pascal). For `dc.w` you can specify any 16-bit relative offset to a local label. For example:

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

ds

`ds[.(b|w|l)]` *size*

Defines a block of *size* bytes, words, or longs. The block is initialized with null characters. If there is no qualifier, `.w` is assumed. For example, this statement defines a block big enough for the structure `DRVRHeader`.

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

entry

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

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

Listing 4.3 Using the entry directive

```
static long MyEntry(void);
static asm long MyFunc(void)
{
    move.l  a,d0
    bra.s   L1

    entry   static MyEntry
    move.l  b,d0
L1: rts
}
```

fralloc

```
fralloc [+]
```

Lets you declare local variables in an assembly function. The `fralloc` directive makes space on the stack for your local stack variables and reserves registers for your local register variables (with the statement `link #x,a6`). For more information, see "Using Local Variables and Arguments in 68K Assembly" on page 119.

There are two versions of `fralloc`. The `fralloc` directive (without a `+`) , pushes modified registers onto the stack. The `fralloc +` directive also pushes all register arguments into their 68K registers.

frfree

```
frfree
```

Frees the stack storage area and restores the registers (with the statement `unlk a6`) that `fralloc` reserved. For more information, see

"Using Local Variables and Arguments in 68K Assembly" on page 119.

machine

```
machine number
```

Specifies which CPU the assembly code is for. The *number* must be one of the following:

`68000`	`68010`	`68020`	`68030`
`68040`	`68349`	`68881`	`68882`
`68851`			

To use the following MC68020 assembler instructions, specify `68020`, `68030`, or `68040`:

`bfchg`	`bfclr`	`bfexts`	`bfextu`
`bfffo`	`bfins`	`bfset`	`bftst`
`divsl`	`divs.l`	`divul`	`divu.l`
`muls.l`	`mulu.l`	`extb.l`	`rtd`

You cannot use MC68020, MC68030, or MC68040 addressing modes. To disable the MC68020 assembler instructions, specify `68000` or `68010`. The arguments `68349`, `68881`, `68882`, and `68851` have no effect.

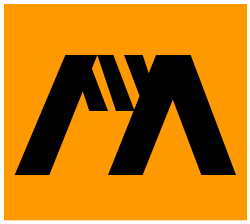
opword

```
opword const-expr (,const-expr)*
```

Lets you include the opcode for an instruction. It works the same as `dc.w`, but emphasizes that the expression is an instruction. For example, this directive calls `WaitNextEvent()`:

```
opword  0xA860      // WaitNextEvent
```

5



PowerPC Assembler Notes

This chapter describes the PowerPC assembler that is part of the CodeWarrior package of compilers.

Overview of PowerPC Assembler Notes

Frequently you want to include a small amount of assembly code in a program. For example, you may want to make sure that a frequently-used function is written as efficiently as possible. Both the PowerPC and 68K compilers include built-in assemblers that let you do just that.

This chapter describes how to use the built-in PowerPC assembler, including its syntax and special directives. It does not document all the instructions available in PowerPC assembler. For more information on the PowerPC programming model, see the IBM *PowerPC User Instruction Set Architecture*. For more information on a particular PowerPC processor and its instruction set, refer to the appropriate document such as the Motorola *PowerPC 601 RISC Microprocessor User's Manual*. The Apple *Assembler for PowerPC* for the MPW s `PPCAsm` assembler is also a good reference and is on the CodeWarrior CD.

TIP: For more information on the built-in 68K assembler, see "Overview of 68K Assembler Notes" on page 115.

The sections in this chapter include:

Writing an Assembly Function for PowerPC

This section details how to write a function for the PowerPC assembler. The topics in this section include:

Defining a Function for PowerPC Assembly

To include assembly in your PowerPC project, declare a function with the `asm` qualifier, like this:

```
asm long f(void) { . . . }
    // OK: An assembly function
```

Note that you cannot create an assembly statement block within a C function:

```
long f(void)
{
  asm { . . . }    // ERROR: Assembly statement
}                  // blocks are not supported.
```

The built-in assembler uses all the standard PowerPC assembler instructions. It accepts some additional directives described in "PowerPC Assembler Directives" on page 134. If you use the `machine` directive, you can also use instructions that are available only in certain versions of the PowerPC. For more information, see "machine" on page 136.

Keep these tips in mind as you write assembly functions:

- All statements must follow this syntax:

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

  Each instruction must end with a newline or a semicolon (;).
- Hex constants must be in C-style , not Pascal-style. For example:

```
li      r3, 0xABCDEF      // OK
li    r3, $ABCDEF         // ERROR
```

- Assembler directives, instructions, and registers are case-sensitive and must be in lowercase. For example these two statements are different:

```
add   r2,r3,r4            // ok
ADD   R2,R3,R4            // ERROR
```

- Every assembly function must end in an `blr` statement. The compiler does not add one for you. For example:

```
asm void f(void)
{
    add  r2,r3,r4
}                   // ERROR: No blr statement

asm void g(void)
{
    add  r2,r3,r4
    blr             // OK
}
```

Listing 5.1 shows an example of an assembly function.

Listing 5.1 Creating an assembly function

```
asm void mystrcpy(char *tostr, char *fromstr)
{
    addi   tostr,tostr,-1
    addi   fromstr,fromstr,-1
@1  lbzu   r5,1(fromstr)
    cmpwi  r5,0
    stbu   r5,1(tostr)
    bne    @1
    blr
}
```

The rest of this section describes how to create local variables, access function parameters, refer to fields within a structure, and use the preprocessor with the assembler. A section at the end of the chapter describes some special assembler directives that the built-in assembler allows.

Creating Labels for PowerPC Assembly

A label can be any identifier that you haven't already declared as a local variable. The name may start with `@`, so these are legal names: `foo`, `@foo`, and `@1`. Only labels that don't start with `@` need to end in a colon. For example:

```
asm void foo(void)
{
x1:  add    r3,r4,r5      // OK
@x2: add    r6,r7,r8      // OK
x3   add    r9,r10,r11    // ERROR: Needs colon
@x4  add    r12,r13,r14   // OK
}
```

NOTE: The first statement in an assembly function cannot be a label that starts with @.

Using Comments for Power PCAssembly

You cannot begin comments with a pound sign (#), since the preprocessor uses the pound sign. However, you can use C and C++ comments. For example:

```
add   r3,r4,r5    # ERROR
add   r3,r4,r5    // OK
add   r3,r4,r5    /* OK */
```

Using the Preprocessor for PowerPC Assembly

You can use all preprocessor features, such as comments and macros, in the assembler. However you must end each assembly statement with a semicolon (`;`), since the preprocessor ignores newlines. For example:

```
#define remainder(x,y,z)  \
    divw    z,x,y; \
    mullw   z,z,y; \
    subf   z,z,x
```

Creating a Stack Frame for PowerPC Assembly

You need to create a stack frame for a function, if the function

- Calls other functions
- Uses more than 224 bytes of local variables
- Declares local register variables.

The easiest way to create a stack frame is to use the `fralloc` directive at the beginning of your function and the `frfree` directive just before the `blr` statement. It automatically allocates and deallocates

memory for your local variables and saves and restores the register contents. This example shows where to put these directives:

```
asm void foo ()
{
  fralloc
  // Your code here
  frfree
  blr
}
```

The `fralloc` directive has an optional argument *number* which lets you specify the size in bytes of the parameter area of the stack frame. By default, the compiler creates a 32-byte parameter area. If your assembly-language routine calls any function that takes more than 32 bytes of parameters, you must specify a larger amount.

Using Local Variables and Arguments for PowerPC Assembly

To refer to a memory location, you can use the name of a local variable or argument.

NOTE: You can refer to local variables by name even if a function does not contain the `fralloc` directive. The PowerPC in-line assembler is different from the 68K in-line assembler in this matter.

The rule for assigning arguments to registers or memory depends on whether the function has a stack frame. If function has a stack frame, the in-line assembler assigns:

- Scalar arguments declared `register` to `r13-r31`
- Floating-point arguments declared `register` to `fp14-fp31`
- Other arguments to memory locations
- Scalar locals declared `register` to `r13-r31`
- Floating-point locals declared `register` to `fp14-fp31`
- Other locals to memory locations

If function has no stack frame, the in-line assembler assigns:

- Arguments that are declared `register` and passed in registers to the appropriate register
- Other arguments to memory locations
- All locals to memory locations

NOTE: If there is no stack frame, a function cannot have more than 224 bytes of local variables.

For more information on PowerPC register conventions and argument-passing conventions, see the Apple *Assembler for PowerPC* on the CodeWarrior CD.

Specifying Instructions for PowerPC Assembly

The PowerPC in-line assembler lets you use most of the basic and extended assembly-language instructions described in the various IBM and Motorola PowerPC User's Guides, such as the Motorola *PowerPC 601 RISC Microprocessor User's Manual*. The Apple *Assembler for PowerPC* for the MPW s `PPCAsm` assembler is also a good reference and is on the CodeWarrior CD.

Each instruction statement corresponds to exactly one PowerPC machine code instruction. All instructions are exactly 4 bytes long. Instruction names are case-sensitive and in all lowercase.

To set the branch prediction (`y`) bit for those branch instructions that can use it, use `+` or `-`. For example:

```
@1  bne+   @2    // Predicts branch taken
@2  bne-   @1    // Predicts branch not taken
```

Most integer instructions have four different forms:

- Normal
- Record, which sets register `cr0` to whether the result is less, than, equal to, or greater than zero. This form ends in a period (`"."`).

- Overflow, which sets the `SO` and `OV` bits in the `XER` if the result overflows. This form ends in the letter "`o`".
- Overflow and Record, which sets both registers. This form ends in "`o.`".

```
add   r3,r4,r5 //  Normal add
add.  r3,r4,r5 //  Add with record: sets cr0
addo  r3,r4,r5 //  Add with overflow:sets XER
addo. r3,r4,r5 //  Add with overflow and
               //  record: sets cr0 and XER
```

Some instructions only have a record form (with a period). Make sure to include the period always:

```
andi.  r3,r4,7   //  '.' is not optional here
andis. r3,r4,7   //  Or here
stwcx. r3,r4,r5  //  Or here
```

Specifying Operands for PowerPC Assembly

This section describes how to specify the operands for assembly language instructions.

Using registers

For a register operand, you must use one of the register names of the appropriate kind for the instruction. The register names are case-sensitive. You can also use a symbolic name for an argument or local variable that was assigned to a register.

The general registers are `RTOC`, `SP`, `r` followed by any number from `0` to `31` (`r0`, `r1`, `r2`, . . . `r31`), or `gpr` followed by any number from `0` to `31` (gpr`0`, `gpr1`, `gpr2`, . . . `gpr31`). The floating-point registers are `fp` followed by any number from `0` to `31` (`fp0`, `fp1`, `fp2`, . . . `fp31`) or `f` followed by any number from `0` to `31` (`f0`, `f1`, `f2`, . . . `f31`). The condition registers are `cr` followed by any number from `0` to `7` (`cr0`, `cr1`, `cr2`, . . . `cr7`).

Using labels

For a label operand, you can use the name of a label. For long branches (such as `b` and `bl` instructions) you can also use function

names. For `bla` and `la` instructions, you use absolute addresses For other branches, you must use the name of a label. For example:

```
b    @3    // OK: Branch to local label
b    foo   // OK: Branch to external
           //     function foo
bl    @3   // OK: Call local label
bl    foo  // OK: Call external function foo
bne    foo // ERROR: Short branch outside
           //        function
```

Using variable names as memory locations

Whenever an instruction requires a memory location (such as load instruction, a store instruction, or `la`), you can use a local or global variable name. You can modify local variable names with struct member references, class member references, array subscripts, or constant displacements. For example, all of the following are valid local variable references:

```
lwz  r3,myVar(SP)  // load myVar into r3
la   r3,myVar(SP)  // load address of myVar
                   //  into r3
lwz  r3,myRect.top
lwz  r3,myArray[2](SP)
lwz  r3,myRectArray[2].top
lbz  r3,myRectArray[2].top+1(SP)
```

You can also use a register variable which is a pointer to a struct or class to access a member of the struct. For example:

```
register Rect *p;
lwz  r3,p->top;
```

Global variable names always refer to the TOC pointer for the variable, not to the variable itself, so you cannot modify them:

```
lwz  r3,myGlobalRect(RTOC)
      // load TOC pointer for myGlobalRect
lwz  r4,Rect.top(r3)
      // fetch 'top' field
lwz  r3,myGlobalRect.top(RTOC)
      // nonsensical
```

You use the same method for obtaining the address of a function:

```
lwz  r3,myFunction(RTOC)
      // load TOC-pointer for TVector
      // to myFunction
```

Using immediate operands

For an immediate operand, you can use an integer or enum constant, `sizeof` expression, and any constant expression using any of the C dyadic and monadic arithmetic operators. These expressions follow the same precedence and associativity rules as normal C expressions. The in-line assembler carries out all arithmetic with 32-bit signed integers.

An immediate operand can also be a reference to a member of a struct or class type. You can use any struct or class name from a `typedef` statement, followed by any number of member references. This evaluates to the offset of the member from the start of the struct. For example:

```
lwz   r4,Rect.top(r3)
addi  r6,r6,Rect.left
```

PowerPC Assembler Directives

This section describes some special assembler directives that the Metrowerks built-in assembler accepts. The directives are listed alphabetically.

entry

`entry [ extern | static ]` *name*

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

Listing 5.2 Using the entry directive

```
void __save_fpr_15(void);
void __save_fpr_16(void);
asm void __save_fpr_14(void)
{
    stfd   fp14,-144(SP)
    entry  __save_fpr_15
    stfd   fp15,-136(SP)
    entry  __save_fpr_16
    stfd  fp16,-128(SP)
    // ...
}
```

fralloc

```
fralloc [ number ]
```

Creates a stack frame for a function and reserves registers for your local register variables. You need to create a stack frame, if the function

- Calls other functions
- Uses more than 224 bytes of local variables
- Declares local register variables.

For more information, see "Creating a Stack Frame for PowerPC Assembly" on page 129.

The `fralloc` directive has an optional argument *number* which lets you specify the size in bytes of the parameter area of the stack frame. By default, the compiler creates a 32-byte parameter area. If your assembly-language routine calls any function that takes more than 32 bytes of parameters, you must specify a larger amount.

frfree

```
frfree
```

Frees the stack frame and restores the registers that `fralloc` reserved. For more information, see "Creating a Stack Frame for PowerPC Assembly" on page 129.

NOTE: The `frfree` directive does not generate a `blr` instruction. You must include one explicitly.

machine

```
machine number
```

Specifies which CPU the assembly code is for. The *number* must be one of the following:

```
601          603          604          all
```

If you use `all`, you can use only those instructions that are available on all PowerPC CPUs. If you don't use the `machine` directive, the compiler assumes `all`.

If you use `601`, you can also use the following instructions:

```
abs       abs.      abso      abso.     clcs
div       div.      divo      divo.     doz
doz.      dozo      dozo.     dozi      lscbx
lscbx.    maskg     maskg.    markir    markir.
mul       mul.      mulo      mulo.     nabs
nabs.     nabso     nabso.    rlmi      rlmi.
rrib      rrib.     sle       sle.      sleq
sleq.     sliq      sliq.     slliq     slliq.
sllq      sllq.     slq       slq.      sraig
sraig.    sraq      sraq.     sre       sre.
srea      srea.     sreq      sreq.     sriq
sriq.     srliq     srliq.    srlq      srlq.
srq       srq.      tlbie
```

If you use `603` or `604`, you can also use the following instructions:

```
fres      fres.     frsqrte   frsqrte.  fsel
fsel.     mftb      mftbl     stfiwx    tlbld
tlbli     tlbsync
```

smclass

```
smclass PR | GL
```

Lets you set the class for a function. By default, all functions have class {PR} which means they are normal executable code. If you're writing a glue routine, like the `__ptr_glue` routine that implements calls through function pointers, use `smclass GL` to set the class to {GL}.

You shouldn't need this directive for your own code, but CodeWarrior PowerPC runtime library uses it frequently

PowerPC Assembler Instructions

The following table gives short descriptions of all the instructions that the PowerPC in-line assembler accepts. If an instruction is available only on certain PowerPC CPUs, the CPUs are listed in brackets at the end of the description, like this: [603, 604].

For more information on the PowerPC programming model, see the IBM *PowerPC User Instruction Set Architecture*. For complete information on the instruction set for a particular PowerPC CP, refer to the appropriate document such as the Motorola *PowerPC 601 RISC Microprocessor User's Manual.*

Instruction	Arguments	Description
`abs`	`rD,rA`	Absolute [601]
`abs.`	`rD,rA`	Absolute [601]
`abso`	`rD,rA`	Absolute [601]
`abso.`	`rD,rA`	Absolute [601]
`add`	`rD,rA,rB`	Add
`add.`	`rD,rA,rB`	Add
`addo`	`rD,rA,rB`	Add
`addo.`	`rD,rA,rB`	Add
`addc`	`rD,rA,rB`	Add Carrying
`addc.`	`rD,rA,rB`	Add Carrying
`addco`	`rD,rA,rB`	Add Carrying
`addco.`	`rD,rA,rB`	Add Carrying
`adde`	`rD,rA,rB`	Add Extended
`adde.`	`rD,rA,rB`	Add Extended
`addeo`	`rD,rA,rB`	Add Extended
`addeo.`	`rD,rA,rB`	Add Extended

Instruction	Arguments	Description
addi	rD,rA,SIMM	Add Immediate
addic	rD,rA,SIMM	Add Immediate Carrying
addic.	rD,rA,SIMM	Add Immediate Carrying and Record
addis	rD,rA,SIMM	Add Immediate Shifted
addme	rD,rA	Add to Minus One Extended
addme.	rD,rA	Add to Minus One Extended
addmeo	rD,rA	Add to Minus One Extended
addmeo.	rD,rA	Add to Minus One Extended
addze	rD,rA	Add to Zero Extended
addze.	rD,rA	Add to Zero Extended
addzeo	rD,rA	Add to Zero Extended
addzeo.	rD,rA	Add to Zero Extended
and	rA,rS,rB	AND
and.	rA,rS,rB	AND
andc	rA,rS,rB	AND with Complement
andc.	rA,rS,rB	AND with Complement
andi.	rA,rS,UIMM	AND Immediate
andis.	rA,rS,UIMM	AND Immediate
b	target	Branch
ba	address	Branch Absolute
bc	BO,BI,target	Branch Conditional
bcctr	BO,BI	Branch Conditional to Count Register
bcctrl	BO,BI	Branch Conditional to Count Register and Link
bcl	BO,BI,target	Branch Conditional and Link
bclr	BO,BI	Branch Conditional to Link Register
bclrl	BO,BI	Branch Conditional to Link Register and Link
bctr		Branch to Count Register

Instruction	Arguments	Description
`bctrl`		Branch to Count Register and Link
`bdnz`	`target`	Decrement CTR, branch if CTR non-zero
`bdnzf`	`BI,target`	Decrement CTR, branch if CTR non-zero and condition False
`bdnzfl`	`BI,target`	Decrement CTR, branch if CTR non-zero and condition False and Link
`bdnzflr`	`BI`	Decrement CTR, branch if CTR non-zero and condition False to Link Register
`bdnzflrl`	`BI`	Decrement CTR, branch if CTR non-zero and condition False to Link Register and Link
`bdnzl`	`target`	Decrement CTR, branch if CTR non-zero and Link
`bdnzlr`		Decrement CTR, branch if CTR non-zero to Link Register
`bdnzlrl`		Decrement CTR, branch if CTR non-zero to Link Register and Link
`bdnzt`	`BI,target`	Decrement CTR, branch if CTR non-zero and condition True
`bdnztl`	`BI,target`	Decrement CTR, branch if CTR non-zero and condition True and Link
`bdnztlr`	`BI`	Decrement CTR, branch if CTR non-zero and condition True to Link Register
`bdnztlrl`	`BI`	Decrement CTR, branch if CTR non-zero and condition True to Link Register and Link
`bdz`	`target`	Decrement CTR, branch if CTR zero
`bdzf`	`BI,target`	Decrement CTR, branch if CTR zero and condition False
`bdzfl`	`BI,target`	Decrement CTR, branch if CTR zero and condition False and Link
`bdzflr`	`BI`	Decrement CTR, branch if CTR zero and condition False to Link Register

Instruction	Arguments	Description
`bdzflrl`	`BI`	Decrement CTR, branch if CTR zero and condition False to Link Register and Link
`bdzl`	`target`	Decrement CTR, branch if CTR zero and Link
`bdzlr`		Decrement CTR, branch if CTR zero to Link Register
`bdzlrl`		Decrement CTR, branch if CTR zero to Link Register and Link
`bdzt`	`BI,target`	Decrement CTR, branch if CTR zero and condition True
`bdztl`	`BI,target`	Decrement CTR, branch if CTR zero and condition True and Link
`bdztlr`	`BI`	Decrement CTR, branch if CTR zero and condition True to Link Register
`bdztlrl`	`BI`	Decrement CTR, branch if CTR zero and condition True to Link Register and Link
`beq`	`[crf,]target`	Branch if Equal
`beqctr`	`[crf]`	Branch if Equal to Count Register
`beqctrl`	`[crf]`	Branch if Equal to Count Register and Link
`beql`	`[crf,]target`	Branch if Equal and Link
`beqlr`	`[crf]`	Branch if Equal to Link Register
`beqlrl`	`[crf]`	Branch if Equal to Link Register and Link
`bf`	`BI,target`	Branch if Condition False
`bfctr`	`BI`	Branch if Condition False to Count Register
`bfctrl`	`BI`	Branch if Condition False to Count Register and Link
`bfl`	`BI,target`	Branch if Condition False and Link
`bflr`	`BI`	Branch if Condition False to Link Register
`bflrl`	`BI`	Branch if Condition False to Link Register and Link
`bge`	`[crf,]target`	Branch if Greater or Equal

Instruction	Arguments	Description
`bgectr`	`[crf]`	Branch if Greater or Equal to Count Register
`bgectrl`	`[crf]`	Branch if Greater or Equal to Count Register and Link
`bgel`	`[crf,]target`	Branch if Greater or Equal and Link
`bgelr`	`[crf]`	Branch if Greater or Equal to Link Register
`bgelrl`	`[crf]`	Branch if Greater or Equal to Link Register and Link
`bgt`	`[crf,]target`	Branch if Greater
`bgtctr`	`[crf]`	Branch if Greater to Count Register
`bgtctrl`	`[crf]`	Branch if Greater to Count Register and Link
`bgtl`	`[crf,]target`	Branch if Greater and Link
`bgtlr`	`[crf]`	Branch if Greater to Link Register
`bgtlrl`	`[crf]`	Branch if Greater to Link Register and Link
`bl`	`target`	Branch and Link
`bla`	`address`	Branch and Link Absolute
`ble`	`[crf,]target`	Branch if Less or Equal
`blectr`	`[crf]`	Branch if Less or Equal to Count Register
`blectrl`	`[crf]`	Branch if Less or Equal to Count Register and Link
`blel`	`[crf,]target`	Branch if Less or Equal and Link
`blelr`	`[crf]`	Branch if Less or Equal to Link Register
`blelrl`	`[crf]`	Branch if Less or Equal to Link Register and Link
`blr`		Branch to Link Register
`blrl`		Branch to Link Register and Link
`blt`	`[crf,]target`	Branch if Less
`bltctr`	`[crf]`	Branch if Less to Count Register
`bltctrl`	`[crf]`	Branch if Less to Count Register and Link

Instruction	Arguments	Description
`bltl`	`[crf,]target`	Branch if Less and Link
`bltlr`	`[crf]`	Branch if Less to Link Register
`bltlrl`	`[crf]`	Branch if Less to Link Register and Link
`bne`	`[crf,]target`	Branch if Not Equal
`bnectr`	`[crf]`	Branch if Not Equal to Count Register
`bnectrl`	`[crf]`	Branch if Not Equal to Count Register and Link
`bnel`	`[crf,]target`	Branch if Not Equal and Link
`bnelr`	`[crf]`	Branch if Not Equal to Link Register
`bnelrl`	`[crf]`	Branch if Not Equal to Link Register and Link
`bng`	`[crf,]target`	Branch if Not Greater
`bngctr`	`[crf]`	Branch if Not Greater to Count Register
`bngctrl`	`[crf]`	Branch if Not Greater to Count Register and Link
`bngl`	`[crf,]target`	Branch if Not Greater and Link
`bnglr`	`[crf]`	Branch if Not Greater to Link Register
`bnglrl`	`[crf]`	Branch if Not Greater to Link Register and Link
`bnl`	`[crf,]target`	Branch if Not Less
`bnlctr`	`[crf]`	Branch if Not Less to Count Register
`bnlctrl`	`[crf]`	Branch if Not Less to Count Register and Link
`bnll`	`[crf,]target`	Branch if Not Less and Link
`bnllr`	`[crf]`	Branch if Not Less to Link Register
`bnllrl`	`[crf]`	Branch if Not Less to Link Register and Link
`bns`	`[crf,]target`	Branch if Not Summary Overflow
`bnsctr`	`[crf]`	Branch if Not Summary Overflow to Count Register

Instruction	Arguments	Description
`bnsctrl`	`[crf]`	Branch if Not Summary Overflow to Count Register and Link
`bnsl`	`[crf,]target`	Branch if Not Summary Overflow and Link
`bnslr`	`[crf]`	Branch if Not Summary Overflow to Link Register
`bnslrl`	`[crf]`	Branch if Not Summary Overflow to Link Register and Link
`bnu`	`[crf,]target`	Branch if Not Unordered
`bnuctr`	`[crf]`	Branch if Not Unordered to Count Register
`bnuctrl`	`[crf]`	Branch if Not Unordered to Count Register and Link
`bnul`	`[crf,]target`	Branch if Not Unordered and Link
`bnulr`	`[crf]`	Branch if Not Unordered to Link Register
`bnulrl`	`[crf]`	Branch if Not Unordered to Link Register and Link
`bso`	`[crf,]target`	Branch if Summary Overflow
`bsoctr`	`[crf]`	Branch if Summary Overflow to Count Register
`bsoctrl`	`[crf]`	Branch if Summary Overflow to Count Register and Link
`bsol`	`[crf,]target`	Branch if Summary Overflow and Link
`bsolr`	`[crf]`	Branch if Summary Overflow to Link Register
`bsolrl`	`[crf]`	Branch if Summary Overflow to Link Register and Link
`bt`	`BI,target`	Branch if Condition True
`btctr`	`BI`	Branch if Condition True to Count Register
`btctrl`	`BI`	Branch if Condition True to Count Register and Link
`btl`	`BI,target`	Branch if Condition True and Link
`btlr`	`BI`	Branch if Condition True to Link Register

Instruction	Arguments	Description
`btlrl`	`BI`	Branch if Condition True to Link Register and Link
`bun`	`[crf,]target`	Branch if Unordered
`bunctr`	`[crf]`	Branch if Unordered to Count Register
`bunctrl`	`[crf]`	Branch if Unordered to Count Register and Link
`bunl`	`[crf,]target`	Branch if Unordered to Link Register
`bunlr`	`[crf]`	Branch if Unordered to Link Register and Link
`bunlrl`	`[crf]`	Branch if Unordered and Link
`clcs`	`rD,rA`	Cache Line Compute Size [601]
`cmp`	`crfD,L,rA,rB`	Compare
`cmpi`	`crfD,L,rA,SIMM`	Compare Immediate
`cmpl`	`crfD,L,rA,rB`	Compare Logical
`cmpli`	`crfD,L,rA,UIMM`	Compare Logical Immediate
`cmplw`	`[crfD,]rA,rB`	Compare Logical Word
`cmplwi`	`[crfD,]rA,UIMM`	Compare Logical Word Immediate
`cmpw`	`[crfD,]rA,rB`	Compare Word
`cmpwi`	`[crfD,]rA,SIMM`	Compare Word Immediate
`cntlzw`	`rA,rS`	Count Leading Zeros Word
`crand`	`crbD,crbA,crbB`	Condition Register AND
`crandc`	`crbD,crbA,crbB`	Condition Register AND with Complement
`creqv`	`crbD,crbA,crbB`	Condition Register Equivalent
`crnand`	`crbD,crbA,crbB`	Condition Register NAND
`crnor`	`crbD,crbA,crbB`	Condition Register NOR
`cror`	`crbD,crbA,crbB`	Condition Register OR
`crorc`	`crbD,crbA,crbB`	Condition Register OR with Complement
`crxor`	`crbD,crbA,crbB`	Condition Register XOR

Instruction	Arguments	Description
dcbf	rA,rB	Data Cache Block Flush
dcbi	rA,rB	Data Cache Block Invalidate
dcbst	rA,rB	Data Cache Block Store
dcbt	rA,rB	Data Cache Block Touch
dcbtst	rA,rB	Data Cache Block Touch for Store
dcbz	rA,rB	Data Cache Block Zero
div	rD,rA,rB	Divide [601]
div.	rD,rA,rB	Divide [601]
divo	rD,rA,rB	Divide [601]
divo.	rD,rA,rB	Divide [601]
divs	rD,rA,rB	Divide Short [601]
divs.	rD,rA,rB	Divide Short [601]
divso	rD,rA,rB	Divide Short [601]
divso.	rD,rA,rB	Divide Short [601]
divw	rD,rA,rB	Divide Word
divw.	rD,rA,rB	Divide Word
divwo	rD,rA,rB	Divide Word
divwo.	rD,rA,rB	Divide Word
divwu	rD,rA,rB	Divide Word Unsigned
divwu.	rD,rA,rB	Divide Word Unsigned
divwuo	rD,rA,rB	Divide Word Unsigned
divwuo.	rD,rA,rB	Divide Word Unsigned
doz	rD,rA	Difference or Zero [601]
doz.	rD,rA	Difference or Zero [601]
dozo	rD,rA	Difference or Zero [601]
dozo.	rD,rA	Difference or Zero [601]
dozi	rD,rA,SIMM	Difference or Zero Immediate [601]

Instruction	Arguments	Description
eciwx	rD,rA,rB	External Control Input Word Indexed
ecowx	rD,rA,rB	External Control Output Word Indexed
eieio		Enforce In-Order Execution of I/O
eqv	rA,rS,rB	Equivalent
eqv.	rA,rS,rB	Equivalent
extsb	rA,rS	Extend Sign Byte
extsb.	rA,rS	Extend Sign Byte
extsh	rA,rS	Extend Sign Halfword
extsh.	rA,rS	Extend Sign Halfword
fabs	frD,frB	Floating-Point Absolute Value
fabs.	frD,frB	Floating-Point Absolute Value
fadd	frD,frA,frB	Floating-Point Add
fadd.	frD,frA,frB	Floating-Point Add
fadds	frD,frA,frB	Floating-Point Add Single
fadds.	frD,frA,frB	Floating-Point Add Single
fcmpo	[crfD,]frA,frB	Floating-Point Compare Ordered
fcmpu	[crfD,]frA,frB	Floating-Point Compare Unordered
fctiw	frD,frB	Floating-Point Convert to Integer Word
fctiw.	frD,frB	Floating-Point Convert to Integer Word
fctiwz	frD,frB	Floating-Point Convert to Integer Word with Round toward Zero
fctiwz.	frD,frB	Floating-Point Convert to Integer Word with Round toward Zero
fdiv	frD,frA,frB	Floating-Point Divide
fdiv.	frD,frA,frB	Floating-Point Divide
fdivs	frD,frA,frB	Floating-Point Divide Single
fdivs.	frD,frA,frB	Floating-Point Divide Single

Instruction	Arguments	Description
`fmadd`	`frD,frA,frC,frB`	Floating-Point Multiply-Add
`fmadd.`	`frD,frA,frC,frB`	Floating-Point Multiply-Add
`fmadds`	`frD,frA,frC,frB`	Floating-Point Multiply-Add Single
`fmadds.`	`frD,frA,frC,frB`	Floating-Point Multiply-Add Single
`fmr`	`frD,frB`	Floating-Point Move Register
`fmr.`	`frD,frB`	Floating-Point Move Register
`fmsub`	`frD,frA,frC,frB`	Floating-Point Multiply-Subtract
`fmsub.`	`frD,frA,frC,frB`	Floating-Point Multiply-Subtract
`fmsubs`	`frD,frA,frC,frB`	Floating-Point Multiply-Subtract Single
`fmsubs.`	`frD,frA,frC,frB`	Floating-Point Multiply-Subtract Single
`fmul`	`frD,frA,frC`	Floating-Point Multiply
`fmul.`	`frD,frA,frC`	Floating-Point Multiply
`fmuls`	`frD,frA,frC`	Floating-Point Multiply Single
`fmuls.`	`frD,frA,frC`	Floating-Point Multiply Single
`fnabs`	`frD,frB`	Floating-Point Negative Absolute
`fnabs.`	`frD,frB`	Floating-Point Negative Absolute
`fneg`	`frD,frB`	Floating-Point Negate
`fneg.`	`frD,frB`	Floating-Point Negate
`fnmadd`	`frD,frA,frC,frB`	Floating-Point Negative Multiply-Add
`fnmadd.`	`frD,frA,frC,frB`	Floating-Point Negative Multiply-Add
`fnmadds`	`frD,frA,frC,frB`	Floating-Point Negative Multiply-Add Single
`fnmadds.`	`frD,frA,frC,frB`	Floating-Point Negative Multiply-Add Single
`fnmsub`	`frD,frA,frC,frB`	Floating-Point Negative Multiply-Subtract
`fnmsub.`	`frD,frA,frC,frB`	Floating-Point Negative Multiply-Subtract
`fnmsubs`	`frD,frA,frC,frB`	Floating-Point Negative Multiply-Subtract Single

Instruction	Arguments	Description
`fnmsubs.`	`frD,frA,frC,frB`	Floating-Point Negative Multiply-Subtract Single
`fres`	`frD,frB`	Floating-Point Reciprocal Estimate Single [603, 604]
`fres.`	`frD,frB`	Floating-Point Reciprocal Estimate Single [603, 604]
`frsp`	`frD,frB`	Floating-Point Round to Single Precision
`frsp.`	`frD,frB`	Floating-Point Round to Single Precision
`frsqrte`	`frD,frB`	Floating-Point Reciprocal Square Root Estimate [603, 604]
`frsqrte.`	`frD,frB`	Floating-Point Reciprocal Square Root Estimate [603, 604]
`fsel`	`frD,frA,frC,frB`	Floating-Point Select [603, 604]
`fsel.`	`frD,frA,frC,frB`	Floating-Point Select [603, 604]
`fsub`	`frD,frA,frB`	Floating-Point Subtract
`fsub.`	`frD,frA,frB`	Floating-Point Subtract
`fsubs`	`frD,frA,frB`	Floating-Point Subtract Single
`fsubs.`	`frD,frA,frB`	Floating-Point Subtract Single
`icbi`	`rA,rB`	Instruction Cache Block Invalidate
`isync`		Instruction Synchronize
`la`	`rD,d(rA)`	Load Address
`lbz`	`rD,d(rA)`	Load Byte and Zero
`lbzu`	`rD,d(rA)`	Load Byte and Zero with Update
`lbzux`	`rD,rA,rB`	Load Byte and Zero with Update Indexed
`lbzx`	`rD,rA,rB`	Load Byte and Zero Indexed
`lfd`	`frD,d(rA)`	Load Floating Double
`lfdu`	`frD,d(rA)`	Load Floating Double with Update
`lfdux`	`frD,rA,rB`	Load Floating Double with Update Indexed

Instruction	Arguments	Description
lfdx	frD,rA,rB	Load Floating Double Indexed
lfs	frD,d(rA)	Load Floating Single
lfsu	frD,d(rA)	Load Floating Single with Update
lfsux	frD,rA,rB	Load Floating Single with Update Indexed
lfsx	frD,rA,rB	Load Floating Single Indexed
lha	rD,d(rA)	Load Halfword Algebraic
lhau	rD,d(rA)	Load Halfword Algebraic with Update
lhaux	rD,rA,rB	Load Halfword Algebraic with Update Indexed
lhax	rD,rA,rB	Load Halfword Algebraic Indexed
lhbrx	rD,rA,rB	Load Halfword Byte-Reversed Indexed
lhz	rD,d(rA)	Load Halfword and Zero
lhzu	rD,d(rA)	Load Halfword and Zero with Update
lhzux	rD,rA,rB	Load Halfword and Zero with Update Indexed
lhzx	rD,rA,rB	Load Halfword and Zero Indexed
li	rD,SIMM	Load Immediate
lis	rD,SIMM	Load Immediate Shifted
lmw	rD,d(rA)	Load Multiple Word
lscbx	rD,rA,rB	Load String and Compare Byte Indexed [601]
lscbx.	rD,rA,rB	Load String and Compare Byte Indexed [601]
lswi	rD,rA,NB	Load String Word Immediate
lswx	rD,rA,rB	Load String Word Indexed
lwarx	rD,rA,rB	Load Word and Reserve Indexed
lwbrx	rD,rA,rB	Load Word Byte-Reversed Indexed
lwz	rD,d(rA)	Load Word and Zero
lwzu	rD,d(rA)	Load Word and Zero with Update
lwzux	rD,rA,rB	Load Word and Zero with Update Indexed

Instruction	Arguments	Description
`lwzx`	`rD,rA,rB`	Load Word and Zero Indexed
`maskg`	`rA,rS,rB`	Mask Generate [601]
`maskg.`	`rA,rS,rB`	Mask Generate [601]
`maskir`	`rA,rS,rB`	Mask Insert from Register [601]
`maskir.`	`rA,rS,rB`	Mask Insert from Register [601]
`mcrf`	`crfD,crfS`	Move Condition Register Field
`mcrfs`	`crfD,crfS`	Move to Condition Register from FPSCR
`mcrxr`	`crfD`	Move to Condition Register from XER
`mfcr`	`rD`	Move from Condition Register
`mfctr`	`rD`	Move from Count Register
`mffs`	`frD`	Move from FPSCR Fields
`mffs.`	`frD`	Move from FPSCR Fields
`mflr`	`rD`	Move from Link Register
`mfmsr`	`rD`	Move from Machine State Register
`mfspr`	`rD,SPR`	Move from Special-Purpose Register
`mfsr`	`rD,SR`	Move from Segment Register
`mfsrin`	`rD,rB`	Move from Segment Register Indirect
`mftb`	`rD`	Move from Time Base Lower [603, 604]
`mftbu`	`rD`	Move from Time Base Upper [603, 604]
`mfxer`	`rD`	Move from XER
`mr`	`rA,rS`	Move Register
`mr.`	`rA,rS`	Move Register
`mtcrf`	`CRM,rS`	Move to Condition Register Fields
`mtctr`	`rS`	Move to Count Register
`mtfsb0`	`crbD`	Move to FPSCR Bit 0
`mtfsb0.`	`crbD`	Move to FPSCR Bit 0
`mtfsb1`	`crbD`	Move to FPSCR Bit 1

Instruction	Arguments	Description
mtfsb1.	crbD	Move to FPSCR Bit 1
mtfsf	FM,frB	Move to FPSCR Fields
mtfsf.	FM,frB	Move from FPSCR Fields
mtfsfi	crfD,IMM	Move to FPSCR Field Immediate
mtfsfi.	crfD,IMM	Move to FPSCR Field Immediate
mtlr	rS	Move to Link Register
mtmsr	rS	Move to Machine State Register
mtspr	SPR,rS	Move to Special Purpose Register
mtsr	SR,rS	Move to Segment Register
mtsrin	rS,rB	Move to Segment Register Indirect
mtxer	rS	Move to XER
mul	rD,rA,rB	Multiply [601]
mul.	rD,rA,rB	Multiply [601]
mulo	rD,rA,rB	Multiply [601]
mulo.	rD,rA,rB	Multiply [601]
mulhw	rD,rA,rB	Multiply High Word
mulhw.	rD,rA,rB	Multiply High Word
mulhwu	rD,rA,rB	Multiply High Word Unsigned
mulhwu.	rD,rA,rB	Multiply High Word Unsigned
mulli	rD,rA,SIMM	Multiply Low Immediate
mullw	rD,rA,rB	Multiply Low Word
mullw.	rD,rA,rB	Multiply Low Word
mullwo	rD,rA,rB	Multiply Low Word
mullwo.	rD,rA,rB	Multiply Low Word
nabs	rD,rA	Negative Absolute [601]
nabs.	rD,rA	Negative Absolute [601]
nabso	rD,rA	Negative Absolute [601]

Instruction	Arguments	Description
nabso.	rD,rA	Negative Absolute [601]
nand	rA,rS,rB	NAND
nand.	rA,rS,rB	NAND
neg	rD,rA	Negate
neg.	rD,rA	Negate
nego	rD,rA	Negate
nego.	rD,rA	Negate
nop		No Operation
nor	rA,rS,rB	NOR
nor.	rA,rS,rB	NOR
not	rA,rS	NOT
not.	rA,rS	NOT
or	rA,rS,rB	OR
or.	rA,rS,rB	OR
orc	rA,rS,rB	OR with Complement
orc.	rA,rS,rB	OR with Complement
ori	rA,rS,UIMM	OR Immediate
oris	rA,rS,UIMM	OR Immediate
rfi		Return from Interrupt
rlmi	rA,rS,rB,MB,ME	Rotate Left then Mask Insert [601]
rlmi.	rA,rS,rB,MB,ME	Rotate Left then Mask Insert [601]
rlwimi	rA,rS,SH,MB,ME	Rotate Left Word Immediate then Mask Insert
rlwimi.	rA,rS,SH,MB,ME	Rotate Left Word Immediate then Mask Insert
rlwinm	rA,rS,SH,MB,ME	Rotate Left Word Immediate then AND with Mask
rlwinm.	rA,rS,SH,MB,ME	Rotate Left Word Immediate then AND with Mask

Instruction	Arguments	Description
`rlwnm`	`rA,rS,rB,MB,ME`	Rotate Left Word then AND with Mask
`rlwnm.`	`rA,rS,rB,MB,ME`	Rotate Left Word then AND with Mask
`rrib`	`rA,rS,rB`	Rotate Right and Insert Bit [601]
`rrib.`	`rA,rS,rB`	Rotate Right and Insert Bit [601]
`sc`		System Call
`sle`	`rA,rS,rB`	Shift Left Extended [601]
`sle.`	`rA,rS,rB`	Shift Left Extended [601]
`sleq`	`rA,rS,rB`	Shift Left Extended with MQ [601]
`sleq.`	`rA,rS,rB`	Shift Left Extended with MQ [601]
`sliq`	`rA,rS,SH`	Shift Left Immediate with MQ [601]
`sliq.`	`rA,rS,SH`	Shift Left Immediate with MQ [601]
`slliq`	`rA,rS,SH`	Shift Left Long Immediate with MQ [601]
`slliq.`	`rA,rS,SH`	Shift Left Long Immediate with MQ [601]
`sllq`	`rA,rS,rB`	Shift Left Long with MQ [601]
`sllq.`	`rA,rS,rB`	Shift Left Long with MQ [601]
`slq`	`rA,rS,rB`	Shift Left with MQ [601]
`slq.`	`rA,rS,rB`	Shift Left with MQ [601]
`slw`	`rA,rS,rB`	Shift Left Word
`slw.`	`rA,rS,rB`	Shift Left Word
`sraiq`	`rA,rS,SH`	Shift Right Algebraic Immediate with MQ [601]
`sraiq.`	`rA,rS,SH`	Shift Right Algebraic Immediate with MQ [601]
`sraq`	`rA,rS,rB`	Shift Right Algebraic with MQ [601]
`sraq.`	`rA,rS,rB`	Shift Right Algebraic with MQ [601]
`sraw`	`rA,rS,rB`	Shift Right Algebraic Word
`sraw.`	`rA,rS,rB`	Shift Right Algebraic Word

Instruction	Arguments	Description
srawi	rA,rS,SH	Shift Right Algebraic Word Immediate
srawi.	rA,rS,SH	Shift Right Algebraic Word Immediate
sre	rA,rS,rB	Shift Right Extended [601]
sre.	rA,rS,rB	Shift Right Extended [601]
srea	rA,rS,rB	Shift Right Extended Algebraic [601]
srea.	rA,rS,rB	Shift Right Extended Algebraic [601]
sreq	rA,rS,rB	Shift Right Extended with MQ [601]
sreq.	rA,rS,rB	Shift Right Extended with MQ [601]
sriq	rA,rS,SH	Shift Right Immediate with MQ [601]
sriq.	rA,rS,SH	Shift Right Immediate with MQ [601]
srliq	rA,rS,SH	Shift Right Long Immediate with MQ [601]
srliq.	rA,rS,SH	Shift Right Long Immediate with MQ [601]
srlq	rA,rS,rB	Shift Right Long with MQ [601]
srlq.	rA,rS,rB	Shift Right Long with MQ [601]
srq	rA,rS,rB	Shift Right with MQ [601]
srq.	rA,rS,rB	Shift Right with MQ [601]
srw	rA,rS,rB	Shift Right Word
srw.	rA,rS,rB	Shift Right Word
stb	rS,d(rA)	Store Byte
stbu	rS,d(rA)	Store Byte with Update
stbux	rS,rA,rB	Store Byte with Update Indexed
stbx	rS,rA,rB	Store Byte Indexed
stfd	frS,d(rA)	Store Floating Double
stfdu	frS,d(rA)	Store Floating Double with Update
stfdux	frS,rA,rB	Store Floating Double with Update Indexed
stfdx	frS,rA,rB	Store Floating Double Indexed

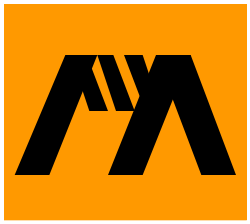
Instruction	Arguments	Description
stfiwx	frS,rA,rB	Store Floating-Point as Integer Word Indexed [603, 604]
stfs	frS,d(rA)	Store Floating Single
stfsu	frS,d(rA)	Store Floating Single with Update
stfsux	frS,rA,rB	Store Floating Single with Update Indexed
stfsx	frS,rA,rB	Store Floating Single Indexed
sth	rS,d(rA)	Store Halfword
sthbrx	rS,rA,rB	Store Halfword Byte-Reversed Indexed
sthu	rS,d(rA)	Store Halfword with Update
sthux	rS,rA,rB	Store Halfword with Update Indexed
sthx	rS,rA,rB	Store Halfword Indexed
stmw	rS,d(rA)	Store Multiple Word
stswi	rS,rA,NB	Store String Word Immediate
stswx	rS,rA,rB	Store String Word Indexed
stw	rS,d(rA)	Store Word
stwbrx	rS,rA,rB	Store Word Byte-Reversed Indexed
stwcx.	rS,rA,rB	Store Word Conditional Indexed
stwu	rS,d(rA)	Store Word with Update
stwux	rS,rA,rB	Store Word with Update Indexed
stwx	rS,rA,rB	Store Word Indexed
sub	rD,rB,rA	Subtract
sub.	rD,rB,rA	Subtract
subo	rD,rB,rA	Subtract
subo.	rD,rB,rA	Subtract
subc	rD,rB,rA	Subtract Carrying
subc.	rD,rB,rA	Subtract Carrying
subco	rD,rB,rA	Subtract Carrying

Instruction	Arguments	Description
subco.	rD,rB,rA	Subtract Carrying
subf	rD,rA,rB	Subtract From
subf.	rD,rA,rB	Subtract From
subfo	rD,rA,rB	Subtract From
subfo.	rD,rA,rB	Subtract From
subfc	rD,rA,rB	Subtract From Carrying
subfc.	rD,rA,rB	Subtract From Carrying
subfco	rD,rA,rB	Subtract From Carrying
subfco.	rD,rA,rB	Subtract From Carrying
subfe	rD,rA,rB	Subtract From Extended
subfe.	rD,rA,rB	Subtract From Extended
subfeo	rD,rA,rB	Subtract From Extended
subfeo.	rD,rA,rB	Subtract From Extended
subfic	rD,rA,SIMM	Subtract From Immediate Carrying
subfme	rD,rA	Subtract From Minus One Extended
subfme.	rD,rA	Subtract From Minus One Extended
subfmeo	rD,rA	Subtract From Minus One Extended
subfmeo.	rD,rA	Subtract From Minus One Extended
subfze	rD,rA	Subtract From Zero Extended
subfze.	rD,rA	Subtract From Zero Extended
subfzeo	rD,rA	Subtract From Zero Extended
subfzeo.	rD,rA	Subtract From Zero Extended
subi	rD,rA,SIMM	Subtract Immediate
subic	rD,rA,SIMM	Subtract Immediate Carrying
subic.	rD,rA,SIMM	Subtract Immediate Carrying and Record
subis	rD,rA,SIMM	Subtract Immediate Shifted
sync		Synchronize

Instruction	Arguments	Description
`tlbie`	`rB`	Translation Lookaside Buffer Invalidate Entry [601, 603]
`tlbld`	`rB`	Load Data TLB Entry [603, 604]
`tlbli`	`rB`	Load Instruction TLB Entry [603, 604]
`tlbsync`		TLB Synchronize [603, 604]
`trap`		Trap Unconditionally
`tw`	`TO,rA,rB`	Trap Word
`tweq`	`rA,rB`	Trap Word Equal
`tweqi`	`rA,SIMM`	Trap Word Equal Immediate
`twge`	`rA,rB`	Trap Word Greater or Equal
`twgei`	`rA,SIMM`	Trap Word Greater or Equal Immediate
`twgt`	`rA,rB`	Trap Word Greater Than
`twgti`	`rA,SIMM`	Trap Word Greater Than Immediate
`twi`	`TO,rA,SIMM`	Trap Word Immediate
`twle`	`rA,rB`	Trap Word Less or Equal
`twlei`	`rA,SIMM`	Trap Word Less or Equal Immediate
`twlge`	`rA,rB`	Trap Word Logical Greater or Equal
`twlgei`	`rA,SIMM`	Trap Word Logical Greater or Equal Immediate
`twlgt`	`rA,rB`	Trap Word Logical Greater Than
`twlgti`	`rA,SIMM`	Trap Word Logical Greater Than Immediate
`twlle`	`rA,rB`	Trap Word Logical Less or Equal
`twllei`	`rA,SIMM`	Trap Word Logical Less or Equal Immediate
`twllt`	`rA,rB`	Trap Word Logical Less Than
`twllti`	`rA,SIMM`	Trap Word Logical Less Than Immediate
`twlng`	`rA,rB`	Trap Word Logical Not Greater
`twlngi`	`rA,SIMM`	Trap Word Logical Not Greater Immediate
`twlnl`	`rA,rB`	Trap Word Logical Not Less

Instruction	Arguments	Description
`twlnli`	`rA,SIMM`	Trap Word Logical Not Less Immediate
`twlt`	`rA,rB`	Trap Word Less Than
`twlti`	`rA,SIMM`	Trap Word Less Than Immediate
`twne`	`rA,rB`	Trap Word Not Equal
`twnei`	`rA,SIMM`	Trap Word Not Equal Immediate
`twng`	`rA,rB`	Trap Word Not Greater
`twngi`	`rA,SIMM`	Trap Word Not Greater Immediate
`twnl`	`rA,rB`	Trap Word Not Less
`twnli`	`rA,SIMM`	Trap Word Not Less Immediate
`xor`	`rA,rS,rB`	XOR
`xor.`	`rA,rS,rB`	XOR
`xori`	`rA,rS,UIMM`	XOR Immediate
`xoris`	`rA,rS,UIMM`	XOR Immediate

6



MIPS Assembler Notes

This chapter describes the MIPS assembler that is part of the Metrowerks C/C++ compiler.

Overview of MIPS Assembler Notes

Frequently you want to include a small amount of assembly code in a program. For example, you may want to make sure that a frequently-used function is written as efficiently as possible. The PowerPC, MIPS, and 68K compilers include built-in assemblers that let you do just that.

This chapter describes how to use the built-in MIPS assembler, including its syntax and special directives. It does not document all the instructions available in MIPS assembler. For more information on the MIPS programming model, see the hardware book for your board.

Writing an Assembly Function

To include assembly in your MIPS project, declare a function with the `asm` qualifier, like this:

```
asm long f(void) { . . . }
    // OK: An assembly function
```

Note that you cannot create an assembly statement block within a C function:

```
long f(void)
{
  asm { . . . }    // ERROR: Assembly statement
}                  // blocks are not supported.
```

The built-in assembler supports all the standard MIPS assembler instructions. It accepts some additional directives described in "Assembler Directive" on page 166, as well as macros.

Keep these tips in mind as you write assembly functions:

- All statements must either be a label, like this:

  [*LocalLabel*:]

  Or an instruction, like this:

  ((*instruction* | *directive*) [*operands*])

  Each statement must end with a newline.
- Hex constants must be in C-style , not Pascal-style. For example:

```
li     t0, 0xABCDEF   // OK
li     t0, $ABCDEF    // ERROR
```

- Assembler directives, instructions, and registers are case-sensitive and must be in lowercase. For example these two statements are different:

```
add   t2,t3,t4        // ok
ADD   T2,T3,T4        // ERROR
```

- Every assembly function must end in an `jr` statement. The compiler does not add one for you. For example:

```
asm void f(void)
{
    add  t2,t3,t4
}                  // ERROR: No jr ra statement
```

```
asm void g(void)
{
    add  t2,t3,t4
    jr   ra        // OK
}
```

- The assembler supports only the three-operand form of the MIPS instructions. For example:

```
add   t0,t1    // ERROR
add   t0,t0,t1 // OK
```

The rest of this section describes how to create local variables, access function parameters, refer to fields within a structure, and use the preprocessor with the assembler. A section at the end of the chapter describes some special assembler directives that the built-in assembler allows.

Creating labels

A label can be any identifier that you haven't already declared as a local variable. A label must end with a colon. An instruction cannot follow a label on the same line. For example:

```
x1:  add    t0,t1,t2     // ERROR
x2:                      // OK
     add    t0,t1,t2     // OK
```

Using comments

You cannot begin comments with a pound sign (#), since the preprocessor uses the pound sign. However, you can begin comments with a semicolon (;) or use C and C++ comments. For example:

```
add   t3,t4,t5    #  ERROR
add   t3,t4,t5    // OK
add   t3,t4,t5    /* OK */
add   t3,t4,t5    ;  OK
```

Using the preprocessor

You can use all preprocessor features, such as comments and macros, in the assembler. However you must end each assembly statement with a semicolon (`;`), since the preprocessor ignores newlines. For example:

```
#define remainder(x,y,z)  \
    div    y,z;           \
    mfhi   x
```

Creating a stack frame

You need to create a stack frame for a function, if the function

- Calls other functions
- Declares local variables.

For more information on creating a stack frame, see the *System V, Application Binary Interface ,MIPS Processor Supplement*.

Specifying operands

This section describes how to specify the operands for assembly language instructions.

Using registers

For a register operand, you must use either:

- The register number with a dollar sign ($) in front

  `$0`, `$1`, `$2`, . . . `$32`
- The software name

  `zero`, `v0`, `v1`, `a0`–`a3`, `t0`–`t9`, `k0`, `k1`, `gp`, `sp`, `fp`, `ra`

  `$f0`–`$f31`

Using parameters

To refer to a parameter, you must use the hardware register it's passed in. For more information on parameter passing, see *System V Application Binary Interface, MIPS Processor Supplement* .

For example:

```
asm int ADD (int x, int y)
{
    // return x + y
    .set   reorder
    add    v0, a0, a1
    jr     ra
}
```

Using global variables

You can refer to global variables by their names. For example:

```
int    Glob;
POINT  P;

asm void INIT (void)
{
    .set    reorder
    sw      zero, Glob
    sw      zero, P.x
    sw      zero, P.y
    jr      ra
}
```

Note that you cannot declare and use local variables in a MIPS assembler function.

Using immediate operands

For an immediate operand, you can use an integer or enum constant, `sizeof` expression, and any constant expression using any of the C dyadic and monadic arithmetic operators. These expressions follow the same precedence and associativity rules as normal C ex-

pressions. The in-line assembler carries out all arithmetic with 32-bit signed integers.

An immediate operand can also be a reference to a member of a struct or class type. You can use any struct or class name from a `typedef` statement, followed by any number of member references. This evaluates to the offset of the member from the start of the struct. For example:

```
lw    t0, Rect.top(a0)
```

Assembler Directive

The MIPS assembler supports one directive.

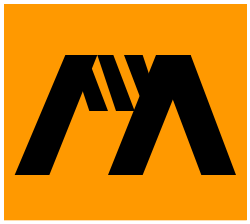
.set

```
.set [ reorder | noreorder ]
```

If you use the `reorder` option, the assembler reorders machine language instructions to improve performance. By default, the assembler uses `noreorder`, and does not reorder instructions. For example:

```
asm int ADD1 (void){
    jr    ra       // return statement
    addi  v0,a0,1  // increment in the branch
                   //   delay slot
}
```

7



Win32/x86 Assembler Notes

This chapter describes the Win32/x86 assembler that is part of the Metrowerks C/C++ compiler.

Overview of Win32/x86 Assembler Notes

Frequently you want to include a small amount of assembly code in a program. For example, you may want to make sure that a frequently-used function is written as efficiently as possible. The PowerPC, 68K, MIPS, and Win32/x86 compilers include built-in assemblers that let you do just that.

This chapter describes how to use the built-in Win32 assembler, including its syntax and special directives. It does not document the instructions available in Win32 assembler.

Writing an Assembly Function

Assembly code for the Win32 compiler uses the following syntax:

```
asm (single_instruction)
```

or

```
asm {multiple_instructions}
```

An `asm` instruction or block may be used wherever a statement is allowed. The assembly instructions are in the standard Intel assembler format.

Assembly instructions may refer to local and global variables as operands. They can use the name of a structure, class, or union as an

immediate operand that evaluates to the size of the structure. To specify the offset of a member, use the structure, union, or class name qualified with the member name, separated by a dot (`.`).

Listing 7.1 shows an example of an assembler function to add two 64-bit integers.

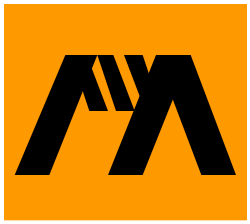
Listing 7.1 Assembler function example

```
struct longlong {int low, high;};

void addlonglong(struct longlong  *a,
                 struct longlong   b)
{
  asm
    {
      mov  eax, a
      mov  ebx, b+longlong.low
      add  longlong.low[eax], ebx
      mov  ebx, b+longlong.high
      adc  longlong.high[eax], ebx
    }
}
```

TIP: If you write a function entirely in assembly language and do not want the standard entry and exit code to be generated, use the declaration modifier `__declspec(naked)`, as described under "Declaration specifiers" on page 27.

WARNING! Although you can mix assembly code and C/C++ code in the same function, you should generally avoid this practice, since it disables all code optimization for that function.

8



Pragmas and Predefined Symbols

This chapter describes the pragmas and predefined symbols available with Metrowerks C/C++.

Overview of Pragmas and Predefined Symbols

This chapter discusses all the pragmas and predefined symbols available with the Metrowerks C/C++ compiler. The sections include:

- Pragmas—lists each pragma
- Predefined Symbols—lists each symbol
- Options Checking—discusses how to check for the state of the compiler

Pragmas

Metrowerks C and C++ let your source code change how the compiler compiles it with pragmas. Most of the pragmas correspond to options in the Project Settings dialog. Usually, you'll use the Preference dialog to set the options for most of your code and use pragmas to change the options for special cases. For example, with the Project Settings dialog, you can turn off a time-consuming optimization and, with a pragma, turn it on only for the code it helps most.

TIP: If you use Metrowerks command-line tools, such as those for MPW or Be OS, see the Command-Line Tools manual for information on how to duplicate the effect of #pragma statements using command-line tool options.

This section includes the following topics:

- Pragma Syntax—how to use pragmas in your code
- The Pragmas—a list of each pragma and its options

Pragma Syntax

Most pragmas have this syntax:

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

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

```
#pragma profile off
  /* If the option Generate Profiler Calls is ,
   * on, turn it off for these functions.
   */
#include <smallfuncs.h>
#pragma profile reset
  /* If the option Generate Profiler Calls was
   * on, turn it back on.
   * Otherwise, the option remains off
   */
```

Suppose that you use `#pragma profile on` instead of `#pragma profile reset`. If you later turn off **Generate Profiler Calls** from the Preference dialog, that pragma turns it on. Using `reset` ensures that you don't inadvertently change the settings in the Project Settings dialog.

The Pragmas

The rest of this section is an alphabetical listing of all pragma options with descriptions.

a6frames (68K Macintosh and Magic Cap)

```
#pragma a6frames on | off | reset
```

If this pragma is on, the compiler generates A6 stack frames which 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 this pragma is off, the compiler does not generate these frames, so the generated code is smaller and faster.

This is the code that the compiler generates for each function, if this pragma is on:

```
LINK #nn,A6
UNLK A6
```

This pragma corresponds to **Generate A6 Stack Frames** option in the 68K Linker settings panel. To check whether this option is on, use `__option (a6frames)`, described in "Options Checking" on page 229.

align (Macintosh and Magic Cap)

```
#pragma options align= alignment
```

This pragma specifies how to align structs and classes, where *alignment* can be one of the following values:

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 Macintosh computers.
`mac68k4byte`	Aligns every field on 4-byte boundaries.

If alignment is	The compiler …
`power`	Align every field on its natural boundary. This is the standard alignment for Power Macintosh computers. 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 Macintosh computers and `power` for Power Macintosh computers.
`packed`	Aligns ever field on a 1-byte boundary. It is not available in any settings panel. This alignment will cause your code to crash or run slowly on many platforms. ***Use it with caution.***
`reset`	Resets the option to the value in the previous `#pragma options align` statement, if there is one, or to the value in the 68K or PPC Processor settings panel.

Note there is a space between `options` and `align`.

This pragma corresponds to the **Struct Alignment** option in the 68K Processor settings panel.

align_array_members (Macintosh and Magic Cap only)

```
#pragma align_array_members on | off | reset
```

This option lets you choose how to align an array in a struct or class. If this option is on, the compiler aligns all array fields larger than a byte according to the setting of the **Struct Alignment** option. If this option is off, the compiler doesn't align array fields.

Listing 8.1 Choosing how to align arrays

```
#pragma align_array_members off
struct X1 {
        char c;         // offset==0
        char arr[4];    // offset==1 (char aligned)
};

#pragma align_array_members on
#pragma align mac68k
struct X2 {
        char c;         // offset==0
        char arr[4];    // offset==2 (2-byte align)
};

#pragma align_array_members on
#pragma align mac68k4byte
struct X3 {
        char c;         // offset==0
        char arr[4];    // offset==4 (4-byte align)
};
```

To check whether this option is on, use `__option(align_array_members)`, described in "Options Checking" on page 229. By default, this option is off.

ANSI_strict

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

The common ANSI extensions are the following. If you turn on the pragma `ANSI_strict`, the compiler generates an error if it encounters any of these extensions.

- C++-style comments. For example:

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

- Unnamed arguments in function definitions. For example:

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

- A # token not followed by an argument in a macro definition. For example:

```
#define add1(x) #x #1
  /* OK, if ANSI_strict is off,
         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`. For example:

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

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

This pragma corresponds to the **ANSI Strict** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (ANSI_strict)`, described in "Options Checking" on page 229.

ARM_conform

```
#pragma ARM_conform on | off | reset
```

When pragma `ARM_conform` is on, the compiler generates an error when it encounters certain ANSI C++ features that conflict with the C++ specification in *The Annotated C++ Reference Manual*. Use this option only if you must make sure that your code strictly follows the specification in *The Annotated C++ Reference Manual.*

Turning on this pragma prevents you from doing the following

- Using protected base classes. For example:

```
class X {};
class Y : protected X {};
  // OK if ARM_conform is off.
```

- Changing the syntax of the conditional operator to let you use assignment expressions without parentheses in the second and third expressions . For example:

```
i ? x=y : y=z
  // OK if ARM_conform is off.
i ? (x=y):(y=z)
  // ALWAYS OK
```

- Declaring variables in the conditions of `if`, `while` and `switch` statements. For example:

```
while (int i=x+y) { . . . }
  // OK if ARM_conform is off.
```

Turning on this option *allows* you to do the following:

- Using variables declared in the condition of an `if` statement after the `if` statement. For example:

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

This pragma corresponds to the **ARM Conformance** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (ARM_conform)`, described in "Options Checking" on page 229.

auto_inline

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

If this pragma is on, the compiler, automatically picks functions to inline for you

Note that if either the **Don't Inline** option ("Inlining functions" on page 52) or the `dont_inline` pragma ("dont_inline" on page 183)

is on, the compiler ignores the setting of the `auto_inline` pragma and doesn't inline any functions.

This pragma corresponds to the **Auto-Inline** option of the **Inlining** menu the C/C++ Language settings panel. To check whether this option is on, use `__option (auto_inline)`, described in "Options Checking" on page 229.

bool (C++ only)

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

When this pragma is on, you can use the standard C++ `bool` type to represent `true` and `false`. Turn this pragma off if recognizing `bool`, `true`, or `false` as keywords would cause problems in your program.

This pragma corresponds to the **Enable bool Support** option in the C/C++ Language settings panel, described in "Using the bool type" on page 96. To check whether this option is on, use `__option(bool)`, described in "Options Checking" on page 229. By default, this option is off.

check_header_flags (precompiled headers only)

```
#pragma check_header_flags on | off | reset
```

When this pragma is `on`, the compiler makes sure 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 project's preferences. If they do not match, the compiler generates an error.

If your precompiled header file has settings that are independent from those in the project, turn this pragma off. If your precompiled header depends on these settings, turn this pragma on.

This pragma does not correspond to any option in the settings panel. To check whether this option is on, use `__option (check_header_flags)`, described in "Options Checking" on page 229. By default, this pragma is off.

code_seg (Win32/x86 only)

```
#pragma code_seg(name)
```

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

```
code_seg(".code")
```

places all subsequent code into a segment named `.code`.

code68020 (68K Macintosh and Magic Cap only)

```
#pragma code68020 on | off | reset
```

When this option is on, the compiler generates code that's optimized for the MC68020. The code runs on a Power Macintosh or a Macintosh with a MC68020 or MC68040. The code does crash on a Macintosh with a MC68000. When this option is off, the compiler generates code that will run on any Macintosh.

WARNING! Do not change this option's setting within a function definition.

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 Macintosh compiler, this option is off by default. In the Magic Cap compiler, this option is on by default. If you change its setting, be sure to change the setting of the pragma `code68349` to the same value.

This pragma corresponds to the **68020 Codegen** option in the 68K Processor settings panel. To check whether this option is on, use `__option (code68020)`, described in "Options Checking" on page 229.

code68349 (Magic Cap only)

```
#pragma code68349 on | off | reset
```

Turning this pragma on automatically turns on the `code68020` pragma as well.

If both this option and the **68020 Codegen** options are on, the compiler does not use certain MC 68020 bitfield instructions which the MC68349 cannot understand, but the compiler does use other MC68020 optimizations. If the **68020 Codegen** option is off, this option has no effect.

In the Macintosh compiler, this option is off by default. In Magic Cap compiler, it's on by default. If you change its setting, be sure to change the setting of the pragma `code68020` to the same value.

This pragma does not correspond to any option in the settings panel. To check whether this option is on, use the `__option(code68349)`, described in "Options Checking" on page 229.

code68881 (68K Macintosh and Magic Cap only)

```
#pragma code68881 on | off | reset
```

When this option is on, the compiler generates code that's optimized for the MC68881 floating-point unit (FPU). This code runs on a Macintosh with an MC68881 FPU, MC68882 FPU, or a MC68040 processor. (The MC68040 has a MC68881 FPU built in.) The code does not run on a Power Macintosh, a Macintosh with an MC68LC040, or a Macintosh with any other processor and no FPU. When this option is off, the compiler generates code that will run on any Macintosh.

WARNING! If you use the `code68881` pragma to turn this option on, place it at the beginning of your file, before you include any files and declare 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 in-

formation on `gestalt()`, see Chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

WARNING! Do not turn this option off in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's off. It is on by default.

This pragma corresponds to the **68881 Codegen** option in the 68K Processor settings panel. To check whether this option is on, use `__option (code68881)`, described in "Options Checking" on page 229.

cplusplus

```
#pragma cplusplus on | off | reset
```

When this pragma is on, the compiler compiles the code that follows as C++ code. When this option is off, the compiler uses the suffix of the filename to determine how to compile it. If a file's name ends in `.cp`, `.cpp`, or `.c++`, the compiler automatically compiles it as C++ code. If a file's name ends in `.c`, the compiler automatically compiles it as C code. You need to use this pragma only if a file contains a mixture of C and C++ code.

This pragma corresponds to the **Activate C++ Compiler** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (cplusplus)`, described in "Options Checking" on page 229.

cpp_extensions

```
#pragma cpp_extensions on | off | reset
```

If this option is on, it enables these extensions to the ANSI C++ standard:

- Anonymous structs. For example:

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

- Unqualified pointer to a member function. For example:

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

  void (Foo::*ptmf2)() = f;
    // OK, if cpp_exptensions is on.
}
```

This pragma does not correspond to any option in the settings panel. To check whether this option is on, use the `__option(cpp_extensions)`, described in "Options Checking" on page 229. By default, this option is off.

d0_pointers (68K Macintosh only)

```
#pragma d0_pointers
```

This pragma lets you choose between two calling conventions: the convention for MPW and Macintosh Toolbox routines and the con-

vention for Metrowerks C and C++ routines. In the MPW and Macintosh Toolbox calling convention, functions return pointers in the register `D0`. In the Metrowerks C and C++ convention, functions return pointers in the register `A0`.

When you declare functions from the Macintosh Toolbox or a library compiled with MPW, turn on the `d0_pointers` pragma. After you declare those functions, turn off the pragma to start declaring or defining Metrowerks C and C++ functions.

In Listing 8.2, the Toolbox functions in `Sound.h` return pointers in `D0` and the user-defined functions in `Myheader.h` use `A0`.

Listing 8.2 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 background 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 (68K Macintosh only)" on page 207.

WARNING! Do not turn this option off in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's off. It is on by default.

This pragma does not correspond to any option in the settings panel. To check whether this option is on, use the `__option(d0_pointers)`, described in "Options Checking" on page 229.

data_seg (Win32/x86 only)

```
#pragma data_seg(name)
```

Ignored. Included for compatibility with Microsoft. It designates the segment into which initialized is placed. The *name* is a string specifying the name of the data segment. For example, the pragma

```
data_seg(".data")
```

places all subsequent data into a segment named `.data`.

direct_destruction (C++ only)

```
#pragma direct_destruction on | off | reset
```

This option is available for backwards-compatibility only and is ignored. Use `#pragma exceptions` instead.

direct_to_som (Macintosh and C++ only)

```
#pragma direct_to_som on | off | reset
```

This pragma lets you create SOM code directly in the CodeWarrior IDE. SOM is an integral part of OpenDoc. For more information, see "Creating Direct-to-SOM Code" on page 105.

Note that when you turn on this program, Metrowerks C/C++ automatically turns on the **Enums Always Int** option in the C/C++ Language settings panel, described in "Enumerated constants of any size" on page 51.

This pragma corresponds to the **Direct to SOM** menu in the C/C++ Language settings panel. Selecting **On** from that menu is like setting this pragma to `on` and setting the `SOMCheckEnviornment` pragma to `off`. Selecting **On with Environment Checks** from that menu is like setting both this pragma and `SOMCheckEnviornment` to on. Selecting **off** from that menu is like setting both this pragma and `SOMCheckEnviornment` to off. For more information on `SOMCheckEnviornment` see "SOMCheckEnvironment (Macintosh and C++ only)" on page 215.

To check whether this option is on, use the `__option (direct_to_SOM)`. See "Options Checking" on page 229. By default, this pragma is off.

disable_registers (PowerPC Macintosh only)

```
#pragma disable_registers on | off | reset
```

If this option is on, the compiler turns off certain optimizations for any function that calls `setjmp()`. It disables global optimization and does not store local variables and arguments in registers. These changes ensures that all locals will have up-to-date values.

NOTE: This option 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()`, but the `try` and `catch` statements are implemented at a lower level and do not use `setjmp()`.

This pragma mimics a feature that's available in THINK C and Symantec C++. Use this pragma only if you're porting code that relies on this feature, since it drastically increases your code's size and decreases its speed. In new code, declare a variable to be `volatile` if you expect its value to persist across `setjmp()` calls.

This pragma does not correspond to any option in the settings panel. To check whether this option is on, use the `__option (disable_registers)`, described in "Options Checking" on page 229. By default, this option is off.

dont_inline

```
#pragma dont_inline on | off | reset
```

If the pragma `dont_inline` is on, the compiler doesn't inline any function calls, even functions declared with the `inline` keyword or member functions defined within a class declaration. Also, it doesn't automatically inline functions, regardless of the setting of the `auto_inline` pragma, described in "auto_inline" on page 175. If this option is off, the compiler expands all inline function calls.

This pragma corresponds to the **Don't Inline** option of the **Inlining** menu the C/C++ Language settings panel. To check whether this option is on, use `__option (dont_inline)`, described in "Options Checking" on page 229.

dont_reuse_strings

```
#pragma dont_reuse_strings on | off | reset
```

If the pragma `dont_reuse_strings` is on, the compiler stores each string literal separately. If this pragma is off, the compiler stores only one copy of identical string literals. This pragma helps you save memory if your program contains lots of identical string literals which you do not modify.

For example, take this code segment:

```
char *str1="Hello";
char *str2="Hello"
*str2 = 'Y';
```

If this option is on, `str1` is `"Hello"` and `str2` is `"Yello"`. If this option is off, both `str1` and `str2` are `"Yello"`.

This pragma corresponds to the **Don't Reuse Strings** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (dont_reuse_strings)`, described in "Options Checking" on page 229.

enumsalwaysints

```
#pragma enumsalwaysint on | off | reset
```

When pragma `enumsalwaysint` is on, the C or C++ compiler makes an enumerated types the same size as an `int`. If an enumerated constant is larger than `int`, the compiler generates an error. When the pragma is off, the compiler makes an enumerated type the size of any integral type. It chooses the integral type with the size that most closely matches the size of the largest enumerated constant. The type could be as small as a `char` or as large as a `long int`.

For example:

```
enum SmallNumber { One = 1, Two = 2 };
  /* If enumsalwaysint is on, this type will
     be the same size as a char.
     If the pragma is off, this type will be
     the same size as an int. */

enum BigNumber
  { ThreeThousandMillion = 3000000000 };
  /* If enumsalwaysint is on, this type will
     be the same size as a long int.
     If this pragma is off, the compiler may
     generate an error. */
```

For more information on how the compiler handles enumerated types, see "Enumerated types" on page 28.

This pragma corresponds to the **Enums Always Int** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (enumsalwaysint)`, described in "Options Checking" on page 229.

exceptions (C++ only)

```
#pragma exceptions on | off | reset
```

If you turn on this pragma, you can use the `try` and `catch` statements to perform exception handling. If your program doesn't use exception handling, turn this option to make your program smaller.

You can throw exceptions across any code that's compiled by the CodeWarrior 8 (or later) Metrowerks C/C++ compiler with the **Enable C++ Exceptions** option turned on. You cannot throw exceptions across the following:

- Macintosh Toolbox function calls
- Libraries compiled with the **Enable C++ Exceptions** option turned off

- Libraries compiled with versions of the Metrowerks C/C++ compiler earlier than CodeWarrior 8
- Libraries compiled with Metrowerks Pascal or other compilers.

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

This pragma corresponds to the **Enable C++ Exceptions** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (exceptions)`, described in "Options Checking" on page 229.

export (Macintosh only)

```
#pragma export on | off | reset | list names
```

The pragma `export` gives you another way to export symbols besides using a `.exp` file. To export symbols with this pragma, choose **Use #pragma** from the **Export Symbols** menu in the PPC PEF or CFM68K settings panel. Then turn on this pragma to export variables and functions declared or defined in this file. If you choose any other option from the **Export Symbols** menu, the compiler ignores this pragma.

If you want to export all the functions and variables declared or defined within a certain range, use `#pragma export on` at the beginning of the range and use `#pragma export off` at the end of the range. If you want to export all the functions and variables in a list, use `#pragma export list`. If you want to export a single variable or function, use `__declspec(export)` at the beginning of the declaration

For example, this code fragment use `#pragma export on` and `off` to export the variable `w` and the functions `a1()` and `b1()`:

```
#pragma export on
int a1(int x, double y);
double b1(int z);
int w;
#pragma export off
```

This code fragment use `#pragma export list` to export the symbols:

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma export list a1, b1, w
```

This code fragment uses `__declspec(internal)` to export the symbols:

```
__declspec(export) int a1(int x, double y);
__declspec(export) double b1(int z);
__declspec(export) int w;
```

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option(export)`, described in "Options Checking" on page 229.

extended_errorcheck

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

If the pragma `extended_errorcheck` is on, the C compiler generates a warning (not an error) if it encounters one of the following:

- A non-`void` function that does not contain a `return` statement. For example, this would generate a warning:

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

This would be OK:

```
void main()
{
  printf ("hello world\n");
}
```

- Assigning an integer or floating-point value to an `enum` type. For example:

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

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

NOTE: Both of these are always errors in C++.

The C and C++ compilers generate a warning if it encounters this:

- An empty `return` statement (`return;`) in a function that is not declared `void`. For example, this code would generate a warning:

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err!=0) return;
      // WARNING: Empty return statement

  // . . .
}
```

This would be OK:

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

  // . . .
}
```

This pragma corresponds to the **Extended Error Checking** option in the C/C++ Warnings settings panel. To check whether this option is

on, use `__option (extended_errorcheck)`, described in "Options Checking" on page 229.

far_code, near_code, smart_code (68K Macintosh and Magic Cap only)

```
#pragma far_code,
#pragma near_code,
#pragma smart_code
```

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 User's Guide*.

These pragmas correspond to the **Code Model** option in the 68K Processor settings panel. The default is `#pragma smart_code`.

far_data (68K Macintosh and Magic Cap only)

```
#pragma far_data on | off | reset
```

If this pragma is on, you can have any amount of global data since the compiler uses 32-bit addressing to refer to globals instead of 16-bit addressing. Your program will also be slightly bigger and slower. this pragma is off, your global data is stored as near data and add to the 64K limit on near data.

This pragma corresponds to the **Far Data** option in the 68K Processor settings panel. To check whether this option is on, use `__option (far_data)`, described in "Options Checking" on page 229.

far_strings (68K Macintosh and Magic Cap only)

```
#pragma far_strings on | off | reset
```

If this pragma is on, you can have any number of string literals since the compiler uses 32-bit addressing to refer to string literals, instead of 16-bit addressing. Your program will also be slightly bigger and slower. If this pragma is off, 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** option in the 68K Processor settings panel. To check whether this option is on, use `__option (far_strings)`, described in "Options Checking" on page 229.

far_vtables (68K Macintosh only)

```
#pragma far_vtables on | off | reset
```

A class with virtual function members has to create a virtual function dispatch table in a data segment. If this pragma is on, that table can be any size since a the compiler uses 32-bit addressing to refer to the table, instead of 16-bit addressing. Your program will also be slightly bigger and slower. If this pragma is off, the table is stored as near data and adds to the 64K limit on near data.

Although the Magic Cap compiler does not raise an error if you use this pragma, it ignores the pragma's value since the Magic Cap compiler does not support C++

This pragma corresponds to the **Far Method Tables** option in the 68K Processor settings panel. To check whether this option is on, use `__option (far_vtables)`, described in "Options Checking" on page 229.

force_active (68K Macintosh only)

```
#pragma force_active on | off | reset
```

If this option is on, the linker will not strip the following functions out of the finished application, even if the functions are never called in the program.

Although the Magic Cap compiler does not raise an error if you use this pragma, it ignores the pragma's value. In Magic Cap code, this option is always on. In Macintosh code, this option is off by default.

This pragma does not correspond to any option in the settings panel. To check whether this option is on, use the `__option (force_active)`, described in "Options Checking" on page 229.

fourbyteints (68K Macintosh only)

```
#pragma fourbyteints on | off | reset
```

When this option is on, the size of an `int` is 4 bytes. When this option is off, the size of an `int` is 2 byes.

WARNING! Do not turn this option off in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's off. It is on by default.

This pragma corresponds to the **4-Byte Ints** option in the 68K Processor settings panel. To check whether this option is on, use `__option (fourbyteints)`, described in "Options Checking" on page 229.

NOTE: Whenever possible, set this option from the settings panel and not from a pragma. If you must set it from a pragma, place the pragma at the beginning of your program, before you include any files or declare any functions or variables.

fp_contract (PowerPC Macintosh only)

```
#pragma fp_contract on | off | reset
```

If this pragma is on, the compiler uses such PowerPC instructions as `FMADD`, `FMSUB`, and `FNMAD` to speed up floating-point computations.

However, certain computations give unexpected results when this pragma is on. For example:

```
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 */
```

When this option is off, `Y` and `Z` have the same value.

This pragma corresponds to the **Use FMADD & FMSUB** option in the PPC Processor settings panel. To check whether this option is on, use `__option (fp_contract)`, described in "Options Checking" on page 229.

function (Win32/x86 only)

```
#pragma function( funcname1, funcname2, ... )
```

Ignored. Included for compatibility with Microsoft.

global_optimizer, optimization_level (PowerPC Macintosh only)

```
#pragma global_optimizer on | off | reset
#pragma optimization_level 1 | 2 | 3 | 4 | 5
```

These pragmas control the global optimizer performs. To turn the global optimizer on and off, use the pragma `global_optimizer`. To choose which optimizations the global optimizer performs, use the pragma `optimization_level` with an argument from `1` to `5`. The higher the argument, the more optimizations that the global optimizer performs. If the global optimizer is turned off, the compiler ignores the pragma `optimization_level`.

Level 1 is the same as the CW4 Global Optimizer. Its optimizations include:

- Register coloring

Level 2 is best for most code. Its optimizations include all those in Level 1 plus these:

- Global common subexpression elimination (also called CSE)
- Copy propagation

Level 3 is best for code with many loops. Its optimizations are all those in Level 2 plus these:

- Moving invariant expressions out of loops (also called Code motion)
- Strength reduction of induction variables
- Using the CTR register for loops that execute a known number of times.
- Loop unrolling.

Level 4 optimizes loops even more, but takes more time. Its options include all those in Level 3 plus this:

- Performing CSE and Code motion a second time, in case the loop optimizations create new opportunities.

These pragmas correspond to the **Global Optimization** option and the **Level** menu in the PPC Processor settings panel. To check whether the global optimizer is on, use `__option (global_optimizer)`, described in "Options Checking" on page 229.

IEEEdoubles (68K Macintosh only)

```
#pragma IEEEdoubles on | off | reset
```

This option, along with the **68881 Codegen** option, specifies the length of a `double`. The table below shows how these options work:

If `IEEEDoubles` **is…**	**and** `code68881` **is…**	**Then a** `double` **is this size…**
on	on or off	64 bits
off	off	80 bits
off	on	96 bits

WARNING! Do not turn this option on in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's on. It is off by default.

This pragma corresponds to the **8-Byte Doubles** option in the 68K Processor settings panel. To check whether this option is on, use `__option (IEEEdoubles)`, described in "Options Checking" on page 229.

NOTE: Whenever possible, set this option from the settings panel and not from a pragma. If you must set it from a pragma, place the pragma at the beginning of your program, before you include any files or declare any functions or variables.

ignore_oldstyle

```
#pragma ignore_oldstyle on | off | reset
```

If pragma `ignore_oldstyle` is on, the compiler ignores old-style function declarations and lets you prototype a function any way you want. In old-style declarations, you don't specify the types of the arguments in the argument list but on separate lines. It's the style of declaration used in the first edition of *The C Programming Language* (Prentice Hall) by Kernighan and Ritchie.

For example, this code defines a prototype for a function with an old-style declaration:

```
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 an option in any settings panel. By default this option is off. To check whether this option is on, use `__option (ignore_oldstyle)`, described in "Options Checking" on page 229.

import (Macintosh only)

`#pragma import on | off | reset | list` *names*

This pragma lets you import variables and functions that are in other fragments. Use this to import symbols that have been exported with the `export` pragma, an `.exp` file, or the **Export Symbols** menu in the CFM68K and PPC PEF settings panel.

If you want to import all the functions and variables declared or define within a certain range, use `#pragma import on` at the beginning of the range and use `#pragma import off` at the end of the range. If you want to import all the functions and variables in a list, use `#pragma import list`. If you want to import a single variable or function, use `__declspec(external)` at the beginning of the declaration

For example, this code fragment use `#pragma import on` and `off` to import the variable `w` and the functions `a1()` and `b1()`:

```
#pragma import on
int a1(int x, double y);
double b1(int z);
int w;
#pragma import off
```

This code fragment use `#pragma import list` to import the symbols:

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma import list a1, b1, w
```

And this code fragment uses `__declspec(import)` to import the symbols:

```
__declspec(import) int a1(int x, double y);
__declspec(import) double b1(int z);
__declspec(import) int w;
```

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (import)`, described in "Options Checking" on page 229.

init_seg (Win32/x86 only)

`pragma init_seg( compiler | lib | user | "`*name* `")`

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

1. `compiler`
2. `lib`
3. `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: it's up to you to call it explicitly.

inline_depth (Win32/x86 only)

`#pragma inline_depth(`*n*`)`

Ignored. Included for compatibility with Microsoft. The number *n* is an integer from `0` to `255`.

internal (Macintosh only)

`#pragma internal on | off | reset | list` *names*

This pragma lets you specify that certain variables and functions are internal and not imported. The compiler generates smaller and faster code when it calls an internal function, even if you declared it as extern.

If you want to declare all the functions and variables declared or define within a certain range as internal, use `#pragma internal on` at the beginning of the range and use `#pragma internal off` at the end of the range. If you want to declare all the functions and variables in a list as internal, use `#pragma internal list`. If you want to declare a single variable or function as internal, use `__declspec(internal)` at the beginning of the declaration.

For example, this code fragment use `#pragma internal on` and `off` to declare the variable `w` and the functions `a1()` and `b1()` as internal:

```
#pragma internal on
int a1(int x, double y);
double b1(int z);
int w;
#pragma internal off
```

This code fragment uses `#pragma internal list` to declare the symbols as internal:

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma internal list a1, b1, w
```

And this code fragment uses `__declspec(internal)` to declare the symbols as internal:

```
__declspec(internal) int a1(int x, double y);
__declspec(internal) double b1(int z);
__declspec(internal) int w;
```

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (internal)`, described in "Options Checking" on page 229.

lib_export (Macintosh only)

```
#pragma lib_export on | off | reset
```

If this pragma is off, the compiler ignores the pragmas export, import, and internal. It is available for compatibility with previous versions of the compiler. It corresponds to the `__declspec(lib_export)` type qualifier, described in "Macintosh and Magic Cap keywords" on page 50. To check whether this option is on, use `__option (lib_export)`, described in "Options Checking" on page 229.

This pragma does not correspond to an option in any settings panel.

longlong

```
#pragma longlong on | off | reset
```

When the `longlong` pragma is on, the C or C++ compiler lets you define a 64-bit integer with the type specifier `long long`. This is twice as large as a long int, which is a 32-bit integer. A `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 an option in any settings panel. To check whether this option is on, use `__option (longlong)`, described in "Options Checking" on page 229.

longlong_enums

```
#pragma longlong_enums on | off | reset
```

This pragma lets you use enumerators that large enough to be `long long` integers. It's ignored if the `enumsalwaysint` pragma is on (described in "enumsalwaysints" on page 184).

For more information on how the compiler handles enumerated types, see "Enumerated types" on page 28.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (longlong_enums)`, described in "Options Checking" on page 229. By default, this option is on.

macsbug, oldstyle_symbols (68K Macintosh and Magic Cap only)

```
#pragma macsbug on | off | reset
#pragma oldstyle_symbols on | off | reset
```

These pragmas 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 turn on 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:

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** option in the 68K Linker settings panel. To check whether the `macsbug` pragma option is on, use `__option (macsbug)`, described in "Options Checking" on page 229. To check whether the old style pragma is on, use `__option (oldstyle_symbols)` described in "Options Checking" on page 229.

mark

```
#pragma mark itemName
```

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 will not appear in the Function pop-up menu.

This pragma does not correspond to an option in any settings panel.

mpwc (68k Macintosh only)

```
#pragma mpwc on | off | reset
```

When the pragma `mpwc` is on, the compiler does the following to be compatible with MPW C's 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 off).
- Returns any 1-byte, 2-byte, or 4-byte structure in `D0`.
- If the **68881 Codegen** option is on, returns any floating-point value in `FP0`.

WARNING! Do not turn this option off in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's off. It is on by default.

This pragma corresponds to the **MPW C Calling Convention** option in the 68K Processor settings panel. To check whether this option is on, use `__option (mpwc)`, described in "Options Checking" on page 229.

mpwc_newline

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

If you turn on the pragma `mpwc_newline`, the compiler uses the MPW conventions for the `'\n'` and `'\r'` characters. If this pragma is off, the compiler uses the Metrowerks C and C++ conventions for these characters.

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

If you want to turn this pragma on, be sure you use the ANSI C and C++ libraries that were compiled with this option on. The 68K versions of these libraries are marked with an N; for example, `ANSI (N/2i) C.68K.Lib`. The PowerPC versions of these libraries are marked with NL; for example, `ANSI (NL) C.PPC.Lib`.

If you turn this pragma on and use the standard ANSI C and C++ libraries, you won't be able to read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` brings you to the beginning of the current line instead of inserting a new line.

WARNING! Do not turn this option off in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's off. It is on by default.

This pragma corresponds to the **Map Newlines to CR** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (mpwc_newline)`, described in "Options Checking" on page 229.

mpwc_relax

```
#pragma mpwc_relax on | off | reset
```

When you turn on this pragma, the compiler treats `char*`, `unsigned char*`, and `Ptr` as the same type. This option is especially useful if you're using code written before the ANSI C standard. This old code frequently used these types interchangeably.

This pragma corresponds to the **Relaxed Pointer Type Rules** option in the C/C++ Language settings panel. To check whether this option is on, `__option (mpwc_relax)`, described in "Options Checking" on page 229.

no_register_coloring (68K Macintosh and Magic Cap only)

```
#pragma no_register_coloring on | off | reset
```

When the `no_register_coloring` pragma is off, 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're using `i` and `j` at the same time and place them in different registers:

```
int k = i + j;
```

If register coloring is on while you debug your project, it may appear as though there's something 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, turn off register coloring or declare the variables you want to watch as volatile.

The pragma corresponds to the **Global Register Allocation** option in the 68K Processor settings panel. To check whether this option is on, use `__option (no_register_coloring)`, described in "Options Checking" on page 229. By default, this option is off.

NOTE: To turn off register coloring in code for a PowerPC Macintosh, use the statement `#pragma global_optimizer off`. For more information, see "global_optimizer, optimization_level (PowerPC Macintosh only)" on page 192.

once

```
#pragma once [ on | off ]
```

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.

This pragma does not correspond to an option in any settings panel. By default this option is off.

oldstyle_symbols (68K Macintosh and Magic Cap only)

See the pragma `macsbug`, described in "macsbug, oldstyle_symbols (68K Macintosh and Magic Cap only)" on page 199.

only_std_keywords

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

The C and C++ compilers recognize additional reserved keywords. If you're writing code that must follow the ANSI standard strictly, turn on the pragma `only_std_keywords`. For more information, see "Additional keywords" on page 50.

This pragma corresponds to the **ANSI Keywords Only** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (only_std_keywords)`, described in "Options Checking" on page 229.

optimization_level (PowerPC Macintosh only)

See the pragma `global_optimizer`, described in "global_optimizer, optimization_level (PowerPC Macintosh only)" on page 192.

optimize_for_size (Macintosh and Magic Cap only)

```
#pragma optimize_for_size on | off | reset
```

This option lets you choose what the compiler does when it must decide between creating small code or fast code. If this option is on, the compiler creates smaller object code at the expense of speed. If this option is off, the compiler creates faster object code at the expense of size.

Most significantly if this option is on, the compiler ignores the `inline` directive, and generates function calls to call any function declared `inline`.

The pragma corresponds to the **Optimize for Size** option in the 68K Processor settings panel and to the **Optimize For** menu in the PPC Processor settings panel. To check whether this option is on, use `__option (optimize_for_size)`, described in "Options Checking" on page 229.

pack (Win32/x86 only)

```
#pragma pack( [n | push, n | pop] )
```

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.

This pragma…	Does this…
`#pragma pack(n)`	Sets the alignment modulus to *n*, where *n* may be `1`, `2`, `4`, `8` or `16`.
`#pragma pack(push, n)`	Pushes the current alignment modulus on a stack, then sets it to *n*, where *n* may 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.
`#pragma pack(pop)`	Pops a previously pushed alignment modulus from the stack.
`#pragma pack()`	Resets alignment modulus to the value specified in the settings panel.

This pragma corresponds to the **Byte Alignment** option in the x86 CodeGen settings panel.

parameter (68K Macintosh and Magic Cap only)

```
#pragma parameter return-reg func-name(param-regs)
```

The compiler passes the parameters for the function *func-name* in the registers specified in *param-regs* instead of the stack, and returns any return 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 "Specifying the registers for arguments (68K Macintosh Only)" on page 45.

This pragma does not correspond to an option in any settings panel.

pcrelstrings (68K Macintosh only)

```
#pragma pcrelstrings on | off | reset
```

If this option is on, the compiler stores the string constants used in a local scope in the code segment and addresses these strings with PC-relative instructions. If this option is off, the compiler stores all string constants in the global data segment. Regardless of how this option is set, the compiler stores string constants used in the global scope in the global data segment. For example:

```
#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 turn the `pool_strings` pragma on, the compiler ignores the setting of the `pcrelstrings` pragma.

WARNING! Do not turn this option off in Magic Cap code. Although the Magic Cap compiler lets you change the setting of this option, your code will not run correctly if it's off. It is on by default.

This pragma corresponds to the **PC-Relative Strings** option in the 68K Processor settings panel. To check whether this option is on, use `__option (pcrelstrings)`, described in "Options Checking" on page 229. By default, this option is off.

peephole (PowerPC Macintosh and Win32/x86 only)

```
#pragma peephole on | off | reset
```

If this pragma is on, 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** option in the PPC Processor settings panel. To check whether this option is on, use `__option (peephole)`, described in "Options Checking" on page 229.

pointers_in_A0, pointers_in_D0 (68K Macintosh only)

```
#pragma pointers_in_A0
#pragma pointers_in_D0
```

These pragmas let you choose between two calling conventions: the convention for MPW and Macintosh Toolbox routines and the convention for Metrowerks C and C++ routines. In the MPW and Macintosh Toolbox calling convention, functions return pointers in the register `D0`. In the Metrowerks C and 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 and C++ functions.

In Listing 8.3, the Toolbox functions in `Sound.h` return pointers in `D0` and the user-defined functions in `Myheader.h` use `A0`.

Listing 8.3 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 (68K Macintosh only)" on page 180.

WARNING! Although the Magic Cap compiler lets you change the settings of these option, your code will not run correctly if `pointers_in_A0` is on and `pointers_in_D0` is off. By default, `pointers_in_A0` is off and `pointers_in_D0` is on.

This pragma does not correspond to any option in the settings panel. To check whether this option is on, use the `__option(d0_pointers)`, described in "Options Checking" on page 229.

pool_strings

```
#pragma pool_strings on | off | reset
```

If the pragma `pool_strings` in the C/C++ Language settings panel is on, the compiler collects all string constants into a single data object so your program needs one TOC entry for all of them. If this pragma is off, the compiler creates a unique data object and TOC entry for each string constant. Turning this pragma on decreases the number of TOC entries in your program but increases

your program's size, since it uses a less efficient method to store the string's address.

This pragma is especially useful if your program is large and has many string constants or uses the Metrowerks Profiler.

NOTE: If you turn the `pool_strings` pragma on, the compiler ignores the setting of the `pcrelstrings` pragma.

This pragma corresponds to the **Pool Strings** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (pool_strings)`, described in "Options Checking" on page 229.

pop, push

```
#pragma push
#pragma pop
```

The pragma `push` saves all the current pragma settings. The pragma `pop` restores all the pragma settings to what they were at the last `push` pragma. For example, see Listing 8.4.

Listing 8.4 push and pop example

```
#pragma far_data on
#pragma pointers_in_A0
#pragma push
   // push all compiler options
#pragma far_data off
#pragma pointers_in_D0
   // pop restores "far_data" and "pointers_in_A0"
#pragma pop
```

These pragmas are available so you can use MacApp with Metrowerks C and C++. If you're writing new code and need to set a pragma option to its original value, use the `reset` argument, described in "Pragma Syntax" on page 170.

precompile_target

```
#pragma precompile_target filename
```

This pragma specifies the filename for a precompiled header file. If you don't specify the filename , the compiler gives the precompiled header file the same name as its source file.

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.

Listing 8.5 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 and C++, 680x0 and PowerPC.

Listing 8.5 Using #pragma precompile_target filename

```
#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
```

profile (Macintosh only)

```
#pragma profile on | off | reset
```

If this pragma is on, 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** option in the 68K Processor settings panel and the Emit Profiler Calls in the PPC Processor settings panel. To check whether this option is on, use `__option (profile)` described in "Options Checking" on page 229.

readonly_strings (PowerPC Macintosh only)

```
#pragma readonly_strings on | off | reset
```

This option determines where to stores string constants. If this option is off, the compiler stores string constants in the data section (class RW). If this option is on, the compiler stores string constants in the code section (class RO).

NOTE: Variables that are not initialized to the address of another object at run time are always placed in the code section (class RO). This includes C/C++ variables declared with the const storage-class modifier.

This pragma corresponds to the **Make Strings ReadOnly** option in the PPC Processor panel. To check whether this option is on, using `#if __option (readonly_strings)`, see "Options Checking" on page 229.

require_prototypes

```
#pragma require_prototypes on | off | reset
```

When the pragma `require_prototypes` is on, 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.

This pragma corresponds to the **Require Function Prototypes** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (require_prototypes)`, described in "Options Checking" on page 229.

RTTI

```
#pragma RTTI on | off | reset
```

When the pragma `RTTI` is on, you can use Run-Time Type Information (or RTTI) features, such as `dyanamic_cast` and `typeid`. The other RTTI expressions are available even if the **Enable RTTI** option is off. Note that `*type_info::before(const type_info&)` is not yet implemented.

This pragma corresponds to the **Enable RTTI** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (RTTI)`, described in "Options Checking" on page 229.

scheduling (PowerPC Macintosh only)

```
#pragma scheduling 601 | 603 | 604 |
                   on | off | reset
```

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.

CodeWarrior lets you choose the type of scheduling that works best for each PowerPC chip. You can use `601`, `603`, or `604`. If you use `on`, the compiler performs 601 scheduling.

However, if you're debugging your code, turn this pragma off. Since it rearranges the instructions produced from your code, the debugger will not be able to match the statements in your source code to the produced instructions.

This pragma corresponds to the **Instruction Scheduling** option in the PPC Processor settings panel.

segment (Macintosh and Magic Cap only)

```
#pragma segment name
```

This pragma places all the functions that follow into the code segment named *name*. For more on function-level segmentation, consult the *CodeWarrior User's Guide*.

Generally, the PowerPC compilers ignore this directive since PowerPC applications do not have code segments. However, if you turn on the **Order Code Sections** option in the PPC PEF settings panel, the PowerPC compilers group functions in the same segment together. For more information, see the *CodeWarrior User's Guide*.

The Magic Cap compiler plugin for the CodeWarrior IDE ignores this pragma and puts all your code in one segment. However, the Magic Cap compiler for MPW does pay attention to this pragma and can segment your code.

This pragma does not correspond to an option in any settings panel.

side_effects (Macintosh only)

```
#pragma side_effects on | off | reset
```

If your program does not contain pointer alias, turn off this pragma to make your program smaller and faster. If your program does use pointer aliases, turn on this pragma to avoid incorrect code. A pointer alias looks like this:

```
int a, *p;
p = &a;   // *p is an alias for a.
```

To understand why pointer aliases are so important, remember that the compiler needs to 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 needs to load `a` into a register again before the second addition, since changing `*p` also changed `a`. If `*p` is not an alias for `a`,

the compiler doesn't need to load `a` into a register again, since changing `*p` does not change `a`.

```
x = a + 1;
*p = 0;      // If *p is an alias for a,
y = a + 2;   //  this changes a.
```

NOTE: The PowerPC compilers ignore this pragma and always assume that a program may contain pointer aliases.

This pragma does not correspond to an option in any settings panel. To check whether this pragma is on, use `__option (side_effects)`, described in "Options Checking" on page 229. By default, this pragma is on.

SOMCallOptimization (Macintosh and C++ only)

```
#pragma SOMCallOptimization on | off | reset
```

The PowerPC compiler uses an optimized error check that is smaller but slightly slower than the one given above. To use the error check show above in PowerPC code, turn this pragma on.

This pragma is ignored if the `direct_to_SOM` pragma, described in "direct_to_som (Macintosh and C++ only)" on page 182, is off.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (SOMCall-Optimization)`. See on "Options Checking" on page 229. By default, this pragma is off.

SOMCallStyle (Macintosh and C++ only)

```
#pragma SOMCallStyle OIDL | IDL
```

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 SOM.

If a class uses the IDL style, its methods must have an Environment pointer as the first parameter. Note that the SOMClass and SOMOb-

ject 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 "Creating Direct-to-SOM Code" on page 105, is off.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (SOMCheckEnvironment)`. See "Options Checking" on page 229. By default, this pragma is set to `IDL`.

SOMCheckEnvironment (Macintosh and C++ only)

```
#pragma SOMCheckEnvironment on | off | reset
```

When the pragma `SOMCheckEnvironment` is on, 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 "Automatic SOM error checking" on page 109.

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 method's result.

And, the compiler transforms this new allocation:

```
new SOMclass;
```

into something that is equivalent to 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 show above in PowerPC code, use the pragma `SOMCall-Optimization`, described in "SOMCallOptimization (Macintosh and C++ only)" on page 214.

This pragma is ignored if the `direct_to_SOM` pragma, described in "Creating Direct-to-SOM Code" on page 105, is off.

This pragma corresponds to the **Direct to SOM** menu in the C/C++ Language settings 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 whether this option is on, use `__option (RTTI)`, described in "Options Checking" on page 229. By default, this pragma is on.

SOMClassVersion (Macintosh and C++ only)

```
#pragma SOMClassVersion(class,majorVer,minorVer)
```

SOM uses the class's version number to make sure the class is compatible with other software you're using. If you don't declare the version numbers, SOM assumes zeroes. The version numbers must be positive or zero.

When you define the class, the program passes its version number to the SOM kernel in the class's 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 "Creating Direct-to-SOM Code" on page 105, is off.

This pragma does not correspond to an option in any settings panel.

SOMMetaClass (Macintosh and C++ only)

```
#pragma SOMMetaClass (class, metaclass)
```

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 "Creating Direct-to-SOM Code" on page 105, is off.

This pragma does not correspond to an option in any settings panel.

SOMReleaseOrder (Macintosh and C++ only)

```
#pragma SOMRelaseOrder(func1, func2,... funcN)
```

A SOM class must specify the release order of its member functions. As a convenience for when you're first developing the class, Metrowerks C++ 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, since 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 inline member functions that are virtual, since they're not considered to be SOM methods. Don't 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 "Creating Direct-to-SOM Code" on page 105, is off.

This pragma does not correspond to an option in any settings panel.

static_inlines

```
#pragma static_inlines on | off | reset
```

The pragma `static_inlines` determines what the compiler does if it cannot inline a call to a function declared `inline` and must create a compiled version of the function. If the pragma is off, the compiler creates one compiled version for the whole project. If the pragma is on, the compiler creates a different compiled version for each file that needs a compiled version.

This pragma is available only so that the compiler can pass certain validation suites. Generally, you'll want to leave this pragma off to make your code smaller without any loss of speed.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (static_inlines)`, described in "Options Checking" on page 229. By default, this pragma is off.

sym

```
#pragma sym on | off | reset
```

The compiler pays attention to this pragma only if you turn on the debug diamond next to the file. If this pragma is off, the compiler does not put debugging information into this source file's SYM file for the functions that follow. If this pragma is on, the compiler does generate debugging information.

Note that the compiler always generates a SYM 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 SYM file.

To check whether this option is on, use `__option (sym)`, described in "Options Checking" on page 229. By default, this pragma is on.

toc_data (PowerPC Macintosh only)

```
#pragma toc_data on | off | reset
```

If the `toc_data` pragma is on, the compiler makes your code smaller and faster. It stores static variables that are 4-bytes long or smaller directly in the TOC, instead of allocating space for them elsewhere and storing pointers to them in the TOC. Turn this pragma off only if your code expects the TOC to contain pointers to data.

This pragma corresponds to the **Store Static Data in TOC** option in the PPC Processor settings panel. To check whether this option is on, use `__option (toc_data)`, described in "Options Checking" on page 229.

trigraphs

```
#pragma trigraphs on | off | reset
```

If you're writing code that must follow the ANSI standard strictly, turn on the pragma `trigraphs` in the C/C++ Language settings panel. Many common Macintosh character constants look like trigraph sequences, and this pragma lets you use them without including escape characters. Be careful when you initialize strings or multi-character constants that contain question marks. For example:

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

This pragma corresponds to the **Expand Trigraphs** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (trigraphs)`, described in "Options Checking" on page 229.

traceback (PowerPC Macintosh only)

```
#pragma traceback on | off | reset
```

This pragma helps other people debug your application or shared library if you do not distribute the source code. If this option is on, the compiler generates an AIX-format traceback table for each function, which are placed in the executable code. Both the Metrowerks and Apple debuggers can use traceback tables.

This pragma corresponds to the **Emit Traceback Tables** option in the PPC Linker settings panel. To check whether this option is on, use the `__option (traceback)`, described in "Options Checking" on page 229. By default, this option is off.

unsigned_char

```
#pragma unsigned_char on | off | reset
```

When the `unsigned_char` pragma is on, the C/C++ compiler treats a `char` declaration as if it were an `unsigned char` declaration.

NOTE: If you turn this pragma on, your code may not be compatible with libraries that were compiled with it turned off. In particular, your code may not work with the ANSI libraries included with CodeWarrior.

This pragma corresponds to the **Use unsigned chars** option in the C/C++ Language settings panel. To check whether this option is on, use `__option (unsigned_char)`, described in "Options Checking" on page 229. By default, this option is off.

unused

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

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 function's scope. You cannot use this pragma

with functions defined within a class definition or with template functions. For example:

```
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int a)
{
  int b;
#pragma unused(a,b)  // Compiler won't complain
                     // that a and b are unused
  // . . .
}
```

This pragma does not correspond to any option in the settings panel.

warn_emptydecl

```
#pragma warn_emptydecl on | off | reset
```

If the pragma `warn_emptydecl` is on, the compiler displays a warning when it encounters a declaration with no variables. For example:

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

This pragma corresponds to the **Empty Declarations** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_emptydecl)`, described in "Options Checking" on page 229.

warning_errors

```
#pragma warning_errors on | off | reset
```

When the pragma `warning_errors` is on, the compiler treats all warnings as though they were errors. It will not compile a file until all warnings are resolved.

This pragma corresponds to the **Treat All Warnings as Errors** option in the C/C++ Warnings settings panel. To check whether this option

is on, use `__option (warning_errors)`, described in "Options Checking" on page 229.

warn_extracomma

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

If the pragma `warn_extracomma` is on, the compiler generates a warning when it encounters an extra comma. For example, this statement is legal in C, but it causes a warning when this pragma is on:

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

This pragma corresponds to the **Treat All Warnings as Errors** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_extracomma)`, described in "Options Checking" on page 229.

warn_hidevirtual

```
#pragma warn_hidevirtual on|off|reset
```

If the pragma `warn_hidevirtual` is on, the compiler generates a warning if you declare a non-virtual member function that hides a virtual function in a superclass. One function hides another if it has the same name but a different argument types. For example:

```
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)
};
```

This pragma corresponds to the **Hidden virtual functions** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_hidevirtual)`. See "Options Checking" on page 229. By default, this option is off.

warn_illpragma

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

If the pragma `warn_illpragma` is on, the compiler displays a warning when it encounters an illegal pragma. For example, these pragma statements generate warnings:

```
#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
```

This pragma corresponds to the **Illegal Pragmas** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_illpragma)`, described in "Options Checking" on page 229.

warn_possunwant

```
#pragma warn_possunwant on | off | reset
```

If the pragma `warn_possunwant` is on, the compiler checks for some common typographical mistakes that are legal C and C++ but that may have unwanted side effects, such as putting in unintended semicolons or confusing = and ==. The compiler generates a warning if it encounters one of these:

- An assignment in a logical expression or the condition in an `if`, `while`, or `for` expression. This check is useful if you frequently use = when you meant to use ==. For example:

```
if (a=b) f();       // WARNING: a=b is an
                    //          assignment

if ((a=b)!=0) f(); // OK: (a=b)!=0 is a
                    //     comparison
```

```
if (a==b) f();     // OK: (a==b) is a
                   //     comparison
```

- An equal comparison in a statement that contains a single expression. This check is useful if you frequently use == when you meant to use =. For example:

```
a == 0;    // WARNING: This is a comparison.
a = 0;     // OK: This is an assignment
```

- A semicolon (;) directly after a `while`, `if`, or `for` statement. For example, the statement generates an error 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 a comment. For example, these statements do not generate errors:

```
while (i++) ;    // OK: White space separation
while (i++)      /* OK: Comment separation */ ;
```

This pragma corresponds to the **Possible Errors** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_possunwant)`, described in "Options Checking" on page 229.

warn_unusedarg

```
#pragma warn_unusedarg on | off | reset
```

If the pragma `warn_unusedarg` is on, the compiler generates a warning when it encounters an argument you declare but do not use. This check helps you find misspelled argument names and arguments you have written out of your program.

```
void foo(int temp, int errer)
{
  error = do_something();  // ERROR: Error is
                           //        undefined
}    // WARNING: temp and error are unused.
```

This pragma corresponds to the **Unused Arguments** option in the C/C++ Warnings settings panel. To check whether this option is on,

use `__option (warn_unusedarg)`, described in "Options Checking" on page 229.

warn_unusedvar

```
#pragma warn_unusedvar on | off | reset
```

If the pragma `warn_unusedvar` is on, the compiler generates a warning when it encounters a variable you declare but do not use. This check helps you find misspelled variable names and variables you have written out of your program. For example:

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

This pragma corresponds to the **Unused Variables** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_unusedvar)`, described in "Options Checking" on page 229.

warning (Win32/x86 only)

```
#pragma warning(warning_specifier : warning_number_list)
```

Ignored. Included for compatibility with Microsoft. The *warning_number_list* is a list of warning numbers separated by spaces, and *warning_specifier* is one of the following:

```
once
default
1
2
3
4
disable
error
```

Predefined Symbols

Metrowerks C and C++ define several preprocessor symbols that give you information about the compile-time environment. Note that these symbols are evaluated at compile time and not at run time.

ANSI Predefined Symbols

The table below lists the symbols that the ANSI C standard requires.

Table 8.1 ANSI predefined symbols

This macro…	is…
`__DATE__`	The date at which the file is compiled; for example, `"Jul 14, 1995"`.
`__FILE__`	The name of the file being compiled; for example `"prog.c"`.
`__LINE__`	The line number of the line being compiled. This is the number before including any header files.
`__TIME__`	The time at which the file is compiled in 24-hour format; for example, `"13:01:45"`.
`__STDC__`	Always `1`. This macro lets you know that Metrowerks C implements the ANSI C standard.

Listing 8.6 shows a small program that uses the ANSI predefined symbols.

Listing 8.6 Using ANSI's 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

The table below lists additional symbols that Metrowerks C and C++ provides.

Table 8.2 Predefined symbols for Metrowerks

This macro…	is…
`__A5__` (68K only.)	`1` if data is A5-relative, `0` if data is A4 relative. It's undefined in the PowerPC compiler.
`__cplusplus`	Defined if you're compiling this file as a C++ file, undefined if you're compiling this file as a C file.
`__fourbyteints__` (68K only.)	`1`, if you turn on the **4-byte Ints** option in the Processor settings panel. `0`, if you turn off that option. It's undefined in the PowerPC compiler.
`__IEEEdoubles__` (68K only.)	`1`, if you turn on the **8-Byte Doubles** option in the Processor settings panel. `0`, if you turn off that option. It's undefined in the PowerPC compiler.
`__INTEL__`	`1`, if you're compiling this code with the Intel compiler. `0`, otherwise.
`__MC68K__`	`1`, if you're compiling this code with the 68K compiler. `0`, otherwise.
`__MC68020__` (68K only.)	`1`, if you turn on the **68020 Codegen** option in the Processor settings panel. `0`, if you turn that option off. It's undefined in the PowerPC compiler.
`__MC68881__` (68K only.)	`1`, if you turn on the **68881 Codegen** option in the Processor settings panel. `0`, if you turn that option off. It's undefined in the PowerPC compiler.
`__MWBROWSER__`	`1`, if the CodeWarrior browser is parsing your code. `0`, otherwise.

This macro…	is…
`__MWERKS__`	The version number of the Metrowerks C/C++ compiler, if you're using CW7 or later. For example, in Metrowerks C/C++ version 7.1 __MWERKS__ would be `0x0710`. It's 1, if you're using an earlier version.
`__profile__`	`1`, if you turn on the **Generate Profiler Calls** option in the Processor settings panel. `0`, if you turn that option off.
`__powerc`, `powerc`, `__POWERPC__`	`1`, if you're compiling this code with the PowerPC compiler. `0`, otherwise.
`macintosh`	`1`, if you're compiling this code with the 68K or PowerPC Macintosh compiler. `0`, otherwise.

Options Checking

The preprocessor function `__option()` lets you test the setting of many pragmas and options in the Project Settings dialog. Its syntax is:

`__option(`*option-name*`)`

If the option is on , `__option ()` returns 1; otherwise, it returns 0.

This function is useful when you want one source file to contains code that's used for different option settings. The example below shows how to compile one series of lines if you're compiling for machines with the MC68881 floating-point unit and another series if you're compiling for machines without out:

```
#if __option (code68881)
  // Code optimized for the floating point unit.
#else
  // Code for any Macintosh
#endif
```

Options table

The table below lists all the option names.

This argument…	Corresponds to the…
`a6frames` (68K only)	**Generate A6 Stack Frames** option in the 68K Linker settings panel and pragma `a6frames`.
`align_array_members`	Pragma `align_array_members`.
`ANSI_strict`	**ANSI Strict** option in the C/C++ Language settings panel and pragma `ANSI_strict`.
`ARM_conform`	**ARM Conformance** option in the C/C++ Language settings panel and pragma `ARM_conform`.
`auto_inline`	**Auto-Inline** option of the **Inlining** menu in the C/C++ Language settings panel and pragma `auto_inline`.
`bool`	Enable **C++ bool/true/false** option in the C/C++ Language settings panel and pragma `bool`.
`check_header_flags`	Pragma `check_header_flags`.
`code68020` (68K only)	**68020 Codegen** option in the 68K Processor settings panel and pragma `code68020`.
`code68881` (68K only)	**68881 Codegen** option in the 68K Processor settings panel and pragma `code68881`.
`code68349` (68K only)	Pragma `code68349`
`cplusplus`	Whether the compiler is compiling this file as a C++ file. Related to the **Activate C++ Compiler** option in the C/C++ Language settings panel, the pragma `cplusplus`, and the macro `cplusplus`

This argument…	Corresponds to the…
`cpp_extensions`	Pragma `cpp_extensions`
`d0_pointers` (68K only)	Pragmas `pointers_in_D0` and `pointers_in_A0`.
`direct_destruction`	**Enable Exception Handling** option in the C/C++ Language settings panel and pragma `direct_destruction.`
`direct_to_SOM`	**Direct to SOM** menu in the C/C++ Language settings panel and pragma `direct_to_SOM`
`disable_registers` (PowerPC only)	Pragma `disable_registers`.
`dont_inline`	**Don't Inline** option in the C/C++ Language settings panel and pragma `dont_inline`.
`dont_reuse_strings`	**Don't Reuse Strings** option in the C/C++ Language settings panel and pragma `dont_reuse_strings.`
`enumsalwaysint`	**Enums Always Int** option in the C/C++ Language settings panel and pragma `enumsalwaysint`
`exceptions`	**Enable C++ Exceptions** option in the C/C++ Language settings panel and pragma `exceptions`
`export`	Pragma `export`.
`extended_errorcheck`	**Extended Error Checking** option in the C/C++ Warnings settings panel and pragma `extended_errorcheck`.
`far_data` (68K only)	**Far Data** option in the 68K Processor settings panel and pragma `far_data.`
`far_strings` (68K only)	**Far String Constants** option in the 68K Processor settings panel and pragma `far_strings`.

This argument…	Corresponds to the…
`far_vtables` (68K only)	Far Method Tables in the 68K Processor settings panel and pragma `far_vtables`.
`force_active` (68K only)	Pragma `force_active`.
`fourbyteints` (68K only)	**4-Byte Ints** option in the 68K Processor settings panel and pragma `fourbyteints`.
`fp_contract` (PowerPC only)	**Use FMADD & FMSUB** option in the PPC Processor settings panel and pragma `fp_contract`.
`global_optimizer` (PowerPC only)	**Global Optimization** option in the PPC Processor settings panel and pragma `global_optimizer`.
`IEEEdoubles` (68K only)	**8-Byte Doubles** option in the 68K Processor settings panel and pragma `IEEEdoubles`.
`ignore_oldstyle`	Pragma `ignore_oldstyle`.
`import`	Pragma `import`.
`internal`	Pragma `internal`.
`lib_export`	Pragma `lib_export`.
`linksym`	a read-only option that is true when the link SYM option in the linker dialog is set
`little_endian`	No option. It is 1 if you're compiling for a little endian target (such as Win32/x86) and 0 if you're compiling for a big endian target (such as Mac OS or Magic Cap).
`longlong`	Pragma `longlong`.
`longlong_enums`	Pragma `longlong_enums`.

This argument…	Corresponds to the…
`macsbug` (68K only)	**MacsBug Symbols** option in the 68K Linker settings panel and pragma `macsbug`.
`mpwc` (68K only)	**MPW C Calling Conventions** option in the 68K Processor settings panel and pragma `mpwc`.
`mpwc_newline`	**Map Newlines to CR** option in the C/C++ Language settings panel and pragma `mpwc_newline`.
`mpwc_relax`	**Relaxed Pointer Type Rules** option in the C/C++ Language settings panel and pragma `mpwc_relax`.
`no_register_coloring`	**Global Register Allocation** option in the 68K Processor settings panel and pragma `no_register_coloring`.
`oldstyle_symbols` (68K only)	**MacsBug Symbols** option in the 68K Linker settings panel and pragma `oldstyle_symbols`
`only_std_keywords`	**ANSI Keywords Only** option in the C/C++ Language settings panel and pragma `only_std_keywords.`
`optimize_for_size`	This corresponds to the **Optimize For Size** option in the 68K Processor settings panel and to the **Optimize For** menu in the PPC Processor settings panel. Also corresponds to the pragma `optimize_for_size`.
`pcrelstrings` (68K only)	**PC-Relative Strings** option in the 68K Processor settings panel and pragma `pcrelstrings`.
`peephole`	**Peephole Optimization** option in the PPC Processor settings panel and pragma `peephole`.

This argument…	Corresponds to the…
`pool_strings`	**Pool Strings** option in the C/C++ Language settings panel and pragma `pool_strings`
`precompile`	Whether the file is being pre-compiled.
`preprocess`	Whether the file is being pre-processed
`profile`	**Generate Profiler Calls** option in the 68K Processor settings panel, **Emit Profiler Calls** option in the PPC Processor settings panel, and pragma `profile`.
`readonly_strings` (PowerPC only)	**Make String Literals Readonly** option in the PPC Processor settings panel and pragma `readonly_strings.`
`require_prototypes`	**Require Function Prototypes** option in the C/C++ Language settings panel and pragma `require_prototypes`.
`RTTI`	**Enable RTTI** option in the C/C++ Language settings panel and pragma `RTTI`.
`side_effects`	Pragma `side_effects`.
`SOMCallOptimization`	Pragma `SOMCallOptimization`
`SOMCheckEnvironment`	**Direct to SOM** menu in the C/C++ Language settings panel and pragma `SOMCheckEnvironment`
`static_inlines`	Pragma `static_inlines`
`sym`	**Generate SYM Files** option in the 68K and PPC Linker settings panels and pragma `sym`

This argument…	Corresponds to the…
`toc_data`	**Store Static Data in TOC** option in the PPC Processor settings panel and pragma `toc_data`
`traceback` (PowerPC only)	Pragma `traceback`.
`trigraphs`	**Expand Trigraphs** option in the C/C++ Language settings panel and pragma `trigraphs.`
`unsigned_char`	**Use Unsigned Chars** option in the C/C++ Language settings panel and pragma `unsigned_char`.
`warn_emptydecl`	**Empty Declarations** option in the C/C++ Warnings settings panel and pragma `warn_emptydecl`.
`warn_extracomma`	**Extra Commas** option in the C/C++ Warnings settings panel and pragma `warn_extracomma`.
`warn_hidevirtual`	**Hidden virtual functions** option in the C/++ Warnings settings panel and pragma `warn_hidevirtual`.
`warn_illpragma`	**Illegal Pragmas** option in the C/C++ Warnings settings panel and pragma `warn_illpragma`.
`warn_possunwant`	**Possible Errors** option in the C/C++ Warnings settings panel and pragma `warn_possunwant`.
`warn_unusedarg`	**Unused Arguments** option in the C/C++ Warnings settings panel and pragma `warn_unusedarg`.

This argument…	Corresponds to the…
`warn_unusedvar`	**Unused Variables** option in the C/C++ Warnings settings panel and pragma `warn_unusedvar`.
`warning_errors`	**Treat Warnings As Errors** option in the C/C++ Warnings settings panel and pragma `warning_errors`.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

CodeWarrior

C, C++, and Assembly Language Reference

Credits

engineering: Andreas Hommel, John McEnerney, Jason Eckhardt

writing: Jeff Mattson

frontline warriors: Fred Peterson

Guide to CodeWarrior Documentation

If you need information about...	See this
Installing updates to CodeWarrior	QuickStart Guide
Getting started using CodeWarrior	QuickStart Guide; Tutorials (Apple Guide)
Using CodeWarrior IDE (Integrated Development Environment)	IDE User's Guide
Debugging	Debugger Manual
Important last-minute information on new features and changes	Release Notes folder
Creating Macintosh and Power Macintosh software	Targeting Mac OS; Mac OS folder
Creating Microsoft Win32/x86 software	Targeting Win32; Win32/x86 folder
Creating Java software	Targeting Java Sun Java Documentation folder
Creating Magic Cap software	Targeting Magic Cap; Magic Cap folder
Using ToolServer with the CodeWarrior editor	IDE User's Guide
Controlling CodeWarrior through AppleScript	IDE User's Guide
Using CodeWarrior to program in MPW	Command Line Tools Manual
C, C++, or 68K assembly-language programming	C, C++, and Assembly Reference; MSL C Reference; MSL C++ Reference
Pascal or Object Pascal programming	Pascal Language Manual; Pascal Library Reference
Fixing compiler and linker errors	Errors Reference
Fixing memory bugs	ZoneRanger Manual
Speeding up your programs	Profiler Manual
PowerPlant	The PowerPlant Book; PowerPlant Advanced Topics; PowerPlant reference documents
Creating a PowerPlant visual interface	Constructor Manual
Creating a Java visual interface	Constructor for Java Manual
Learning how to program for the Mac OS	Discover Programming for Macintosh
Learning how to program in Java	Discover Programming for Java
Contacting Metrowerks about registration, sales, and licensing	Quick Start Guide
Contacting Metrowerks about problems and suggestions using CodeWarrior software	email Report Forms in the Release Notes folder
Sample programs and examples	CodeWarrior Examples folder; The PowerPlant Book; PowerPlant Advanced Topics; Tutorials (Apple Guide)
Problems other CodeWarrior users have solved	Internet newsgroup [docs] folder

961218-1821