CodeWarrior®

Porting 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.

Revised: 99/02/09 map

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`cw_support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	Go: `Metrowerks`

Table of Contents

1

Welcome

Welcome to the *CodeWarrior Porting Reference*. This guide offers hints and tips on moving your programming project from other software development environments to CodeWarrior. It also points you to other CodeWarrior documentation for more in-depth topics.

Use this guide if you are new CodeWarrior or you are converting a programming project from another development system to CodeWarrior.

This guide refers to development tools that might not be available with the CodeWarrior package you have. Consult the *QuickStart* guide that came with your CodeWarrior package for information on what is in your CodeWarrior package.

This chapter has these sections:

Read the Release Notes

Before using CodeWarrior, read the information in the `Release Notes` folder, which is on the CodeWarrior CD-ROM and installed in the CodeWarrior folder on your computer's hard drive.

The release notes contain important information about new features, bug fixes, and any late-breaking changes.

CodeWarrior Year 2000 Compliance

The Products provided by Metrowerks under the License agreement process dates only to the extent that the Products use date data provided by the host or target operating system for date representations used in internal processes, such as file modifications. Any Year 2000 Compliance issues resulting from the operation of the Products are therefore necessarily subject to the Year 2000 Compliance of the relevant host or target operating system. Metrowerks directs you to the relevant statements of Microsoft Corporation, Sun Microsystems, Inc., Apple Computer, Inc., and other host or target operating systems relating to the Year 2000 Compliance of their operating systems. Except as expressly described above, the Products, in themselves, do not process date data and therefore do not implicate Year 2000 Compliance issues.

For additional information, visit: `http://www.metrowerks.com/about/y2k.html`.

What is New in This Release

This guide has been significantly rewritten. It now has new and updated information on moving a programming project to CodeWarrior.

Also, future revisions of this guide will add information about moving to CodeWarrior from more software development environments.

What is in This Guide

The chapters in this guide are listed in Table 1.1.

Table 1.1 Chapters in this guide

Read this chapter…	to learn about
"Welcome" on page 5	using this guide
"Common Porting Issues" on page 9	solutions to common problems you might encounter when porting your project to CodeWarrior
"Microsoft® Visual Studio® Porting Issues" on page 17	converting a programming project from Miscrosoft Visual Studio to CodeWarrior
"THINK® Pascal and MPW Pascal Porting Issues" on page 25	converting a Mac OS programming project from THINK Pascal or MPW Pascal to CodeWarrior Pascal

Where to Go From Here

This guide only discusses unique issues and problems that you might encounter when converting your programming project to CodeWarrior. To learn more about CodeWarrior's tools and libraries, refer to the table listed at the end of this guide, "Guide to CodeWarrior Documentation" on page 38..

If you are new to CodeWarrior, you will find these manuals and references the most useful:

- *CodeWarrior IDE User Guide*—how to use the CodeWarrior IDE to edit, search, navigate, compile, link, debug, and manage programming projects
- *C Compilers Reference*—discusses the CodeWarrior implementations of the C, C++, Embedded C++, and Objective C programming languages
- *MSL C++ Reference*—describes the Metrowerks Standard Library for C
- *MSL C Reference*—describes the Metrowerks Standard Library for C++

- *Pascal Compilers Reference*—discusses the CodeWarrior implementation of the Pascal and Object Pascal programming languages
- *Pascal Language Reference*—describes the elements of the CodeWarrior Pascal programming language
- *Pascal Library Reference*—describes the built-in library routines in CodeWarrior Pascal
- *Targeting* manuals—describe how to use CodeWarrior to develop software for a specific processor or operating system

  For example, to learn how to use CodeWarrior to create software for the Apple Macintosh, read *Targeting Mac OS*.

2

Common Porting Issues

This chapter covers general problems and differences among other software development tools and CodeWarrior. The topics in this chapter describe the CodeWarrior features that have subtle and obvious differences from most other development environments.

This chapter has these sections:

C and C++ Issues

This section covers topics on differences between the CodeWarrior C and C++ compilers and other compilers:

About Metrowerks Standard Libraries

The Metrowerks Standard Libraries (MSL) for C and C++ comply with the libraries described by the ANSI/ISO C and C++ Standards.

MSL C and MSL C++ also have functions that are commonly available but are not part of their respective standards. For example, MSL C has functions that are commonly available in UNIX.

ARM and Other C++ Implementations

CodeWarrior C++ conforms to the ANSI/ISO C++ standard, although it has options to compile source code that conforms to the C++ specification in the *Annotated C++ Reference Manual* (ARM).

For information on compiling non-ANSI C++ source code, refer to the *C, C++, And Assembly Language Reference*.

CFront and Metrowerks C++

MSL C++ conforms to the ANSI/ISO C++ Standard for the C++ streams libraries. There are a few variations from the standard as accepted and published in such books as Stroustrup's *The C++ Programming Language, 2nd edition* (Addison-Wesly, 1991) and Stanley B. Lippman's *C++ Primer, 2nd edition* (Addison-Wesley, 1991) commonly referred to as the C++ Programming Language or CFront C++.

For information on compiling non-ANSI C++ source code, refer to the *C Compilers Reference*.

#include header naming conventions

ANSI C++ no longer require a file name extension for the include files. However, the file extension naming conventions are provided for by ANSI C++, and included in Metrowerks.

CFront C++

```
#include <iostream.h>
```

ANSI C++

```
#include <iostream>
```

File open modes

Listing 2.1 shows the valid combinations of `ios::openmode` for opening a file as defined in the ANSI C++ library—these are defined in terms of the equivalent `modestr` used in `fopen(s, modestr)` (see para 7.9.5.3 of ANSI Standard for C).

Listing 2.1 File open modes

```
ios::in   modestr = "r"
ios::out | ios::trunc  modestr = "w"
ios::out | ios::app  modestr = "a"
ios::in  | ios::bin  modestr = "rb"
ios::out | ios::trunc | ios::bin modestr = "wb"
ios::out | ios::app   | ios::bin modestr = "ab"
ios::in  | ios::out     modestr = "r+"
ios::in  | ios::out   | ios::trunc modestr = "w+"
ios::in  | ios::out   | ios::app modestr = "a+"
ios::in  | ios::out   | ios::bin modestr = "r+b"
ios::in  | ios::out   | ios::trunc | ios::bin modestr = "w+b"
ios::in  | ios::out   | ios::app   | ios::bin modestr = "a+b"
```

All other combinations are invalid and no file is opened and no error message is produced.

File open and close testing

Use the `is_open()` function to test for an open file.

In CFront C++:

```
ofstream to("testFile");
  if (!to ) /* ... */
```

In ANSI C++:

```
ofstream to("testFile");
  if (to.is_open() == 0) /* ... */
```

iostream `fail()` vs. `eof()`

Historically (before the advent of the ANSI/ISO C++ specification) the `eofbit` was set haphazardly. ANSI C++ libraries do not set the `eofbit`, therefore the previous practice of using the function `eof()`, which gave a non-zero result when the end of file was reached, is no longer useful. Instead you should test the value of the `fail()` function, which will pick up both `eof` and other kinds of failure, as shown in Listing 2.2.

Listing 2.2 fail() vs. eof()

```
#include <iostream>
#include <fstream>

void main()
{
  // fill file with 10 x's
         ofstream out("testfile");
         for(int i = 0; i < 10; i++ ) out.put( 'X');
         out.close();

        char c = 0;
        ifstream input("testfile");

    //  while ( !input.fail() )   this leaves EOF character
    //  while( input.peek() != EOF )this works but less safe
  while ( !input.fail() && input.peek() != EOF )
        {
           c = input.get() ;
           cout << "char: " << c << endl;
       }
     input.close();
}
```

UNIX and POSIX Libraries

There are header files that provide some of the functions in the POSIX standard and many functions found in UNIX libraries that are not specified in the ANSI C or ANSI C++ standards.

Refer to `fnctl.h`, `stat.h`, `unistd.h`, `unix.h`, `utime.h`, and `utsname.h` in the *C Library Reference* for more information.

The `sizeof()` operator and `int`

When programming in C, do not assume that the value of a `sizeof()` operator is assignment compatible with the `int` or `long` data types.

According to the ANSI C standard `sizeof()` returns a value of type `size_t`, defined in `stddef.h`.

The `size_t` and `int` data types are often the same, but might differ depending on the platform or processor. Refer to Listing 2.3 for an example of how a program executes incorrectly when it assumes that `size_t` and `int` are the same size.

Listing 2.3 Making assumptions about `sizeof()` and `int`

```
#include <stdio.h>
#include <stddef.h>

typedef struct {
  charbigArray[100000];
} MyStruct;

void main(void)
{
  int j;
  size_t t;


  j = sizeof(MyStruct); /* This doesn't work */
  t = sizeof(MyStruct); /* This works */

  printf("bad size of MyStruct = %ld\n", j);
  printf("good size of  MyStruct = %ld\n", t);
}

/* Output, running on Mac OS for 68K:

bad size of MyStruct = -2036334591
good size of  MyStruct = 100000

*/
```

There are a few variations from the MSL implementation of C++ and *previous* versions of ANSI/ISO C++. MSL C++ conforms as closely as possible to the ANSI/ISO C++ standard.

- The `fstream` class is now included for mixed input and output.
- The file reading facilities `tellg()`, `tellp()`, `seekg()`, and `seekp()` are now included in the ANSI C++ standard.
- The STL algorithms are now part of the proposed standard and conflict with older versions of the STL Libraries.

Mac OS Issues

The topics in this section deal with common problems you might have when porting your programming project to Apple's Mac OS. Of course, this section cannot cover every aspect of porting a project to a new operating system. Instead, it gives you tips and references to other documentation to help you.

The topics in this section are:

Where to Find More Information About Mac OS

You'll find comprehensive documentation, sample source code, and other resources for Mac OS development at Apple's web site for software developers, `http://www.apple.com/developer/`.

To learn how to use CodeWarrior to develop software for Mac OS, see *Targeting Mac OS*.

Console I/O for Mac OS

For programs that use simple text input/output without calling on any Mac OS-specific features, CodeWarrior provides SIOUX. SIOUX is a Mac OS software package that automatically opens a text window, accepts characters from the keyboard, and handles

menus. SIOUX does all this transparently; you, the programmer, do not have to explicitly invoke SIOUX.

Simply by calling a standard C or C++ function that reads from or writes to the standard input, output, or error files will invoke SIOUX automatically.

Refer to the *MSL C Reference* for information on customizing SIOUX.

Command-Line Arguments for Mac OS

The C/C++ `ccommand()` function provides a dialog box that allows the user to enter text as if it were at the command line. Refer to `ccommand()` in the *MSL C Reference* for more information.

File Redirection for Mac OS

For UNIX-style file redirection, use the C/C++ `ccommand()` function. Refer to ccommand in the *MSL C Reference* for more information.

Including Files In C/C++ on Mac OS

When using CodeWarrior on Mac OS, CodeWarrior C/C++ handles subdirectory names in `#include` directives differently than UNIX compilers. In particular, Mac OS uses the colon, ":", as a directory separator character, not the slash, "/".

For example, when using CodeWarrior on a Mac OS computer, issuing this directive

```
#include "special/datatypes.h"
```

will actually include a file named "`special/datatypes.h`," not the file "`datatypes.h`" in the "special" directory.

Specify Subdirectories for #include Directives

The recommended way to specify a directory for an `#include` directive is to add the directory to the list of access paths in the project's **Access Paths** project settings panel and remove references

to subdirectories in all `#include` directives. Refer to the *CodeWarrior IDE Guide* for more information on access path preferences.

Another way to specify a subdirectory for an `#include` directive is to convert the pathname to a Mac OS pathname. For example

```
#include "special/datatypes.h"
```

becomes

```
#include ":special:datatypes.h"
```

3

Microsoft® Visual Studio® Porting Issues

This chapter covers common problems and issues you will encounter when porting a programming project from Microsoft Visual Studio's C/C++ compilers to CodeWarrior. Specifically, this chapter covers differences and potential problems (and solutions) you will encounter when porting a project originally developed for the Win32/x86 platform.

This chapter has these sections:

C/C++ Compiler Differences

Notable differences between CodeWarrior C and C++ compilers and the Microsoft C and C++ compilers are listed here:

Conforming to the ANSI/ISO C and C++ Standards

When the **ANSI Strict** option in the **C/C++ Language** settings panel is enabled, all non-standard language extensions are disabled. Unless you are writing software that must be portable to any ANSI C/C++ platform, you probably should leave ANSI Strict option turned off.

Refer to the *C Compilers Reference* for complete information on the **ANSI Strict** option.

These language extensions include:

- array sizes of 0 and empty arrays as trailing structure members
- multiple identical `typedef`s
- using a `void` return type for `main()`
- unsigned enumeration types
- bitfields that are not `int`-sized
- anonymous unions
- computed goto labels
- GNU-style temporary initialization casts
- `asm()`-form inline assembly
- unnamed arguments
- pointer to integer conversions
- C++-style single-line comments in C source code ("`//`")
- `long long` constants with "`i64`" suffix
- double constants with a "`d`" suffix
- binary constants with a "`0b`" prefix
- # on line by itself
- ignored tokens after `#else` and `#endif`
- supporting `#warning` and `#ident` preprocessor directives

Relaxed Pointer Type Rules

When you turn on the **Relaxed Pointer Type Rules** option in the **C/C++ Language** settings panel, CodeWarrior C overlooks the normal type checking it does when assigning from one pointer to another. The compiler does not warn or give an error message in cases where you mix pointers to different types. For example, when this option is on, the compiler will allow code like this:

```
struct foo *mary;
char *bob = mary;
```

This option should be avoided when possible.

This option has no effect on C++. When compiling C++ source code, the compiler differentiates pointer types even if this option is on.

This option exists to make it easier to port old pre-ANSI C code that assumed that any pointer had the same representation as any other pointer. C source code for Microsoft Windows that uses generic handles (`HANDLE`) instead of specific types of handles (for example, `HWND`) might not compile with this option turned off.

RTTI

CodeWarrior currently does not support Microsoft-compatible C++ runtime type information (RTTI). CodeWarrior does support ANSI/ISO C++ RTTI, but you cannot use `typeof()` or `dynamic_cast<>` on objects compiled by Microsoft C++.

Exception Handling

CodeWarrior provides full support for Microsoft-compatible C++ exception handling.

Name Mangling

CodeWarrior C++'s name mangling on templates is incompatible with Microsoft C++'s mangling when **ARM Conformance** is turned off in the **C/C++ Language** settings panel. With **ARM Conformance**

off, CodeWarrior can support the differentiation between template and non-template functions that do not include the template type in their parameter lists.

For example:

```
template <class T> void foo (int a);
void foo (int a);
```

Using the Microsoft name mangling scheme, these two functions are mangled identically, although, in your code, you could differentiate them by:

```
foo<int> (42);
foo (42);
```

IEEE Floating Point Standards

CodeWarrior C/C++ follows the IEEE standards regarding the outcome of comparison operators and `NaN` (not a number). These are all considered unordered, so any comparison involving `NaN` will be false, except for the not equals comparison (!=) which returns true.

Microsoft C++ version 5 and Microsoft C++ version 6 do not generate instructions to check the unordered flag on comparisons, so the result of the comparison may vary depending on the exact encoding of the `NaN`s involved in the operation.

CodeWarrior C/C++ compilers for x86 and Microsoft C++ correctly handle propagation of `NaN`s through arithmetic operations, as that is handled by the floating point processor. For any operation, if one or both of the operands is a `NaN`, the result will also be a `NaN`.

If you have problems with this discrepancy in your code, you may want to enable the processor exception on `NaN` generation, but most code should never have to deal with these values.

To do so, you can add this code fragment to your project, with a call to `enable_nan_exception()` sometime before you expect the `NaN` to be generated. You should then enable the **Float Invalid Op** exception in the **x86 Exception** debugging settings panel.

```
#include <fenv.h>
void enable_nan_exception(void)
{
   fenv_t fe;
   // this should turn on exceptions
   // on NaN generation
   fegetenv(&fe);
   fe = fe & ~FE_INVALID;
   fesetenv(&fe);
}
```

Inline Assembler

Microsoft uses a syntax for the inline assembler in their C/C++ compiler similar to, but not exactly like MASM. Since this is barely documented, the CodeWarrior assembler attempts to duplicate the observed behavior of the Microsoft assembler. Note that there may be some cases where it will not detect an error.

Some of the inline assembler support includes:

- Both CodeWarrior C++ and Microsoft C++ use Intel syntax for their inline assembler.
- The Microsoft assembler treats labels as case insensitive, while CodeWarrior requires an exact match on case.
- Directives not supported in the x86 assembler: `EVEN`.
- CodeWarrior supports the `SIZE` keyword in assembly expressions for getting the size (in bytes) of an object. CodeWarrior does not support `LENGTH` or `TYPE`.
- CodeWarrior supports the `ALIGN`, `DB`, `DW`, and `DD` directives. CodeWarrior supports `EMIT` rather than `_emit` for directly inserting bytes into the assembly code.
- CodeWarrior ignores the `SHORT` modifier on jump instructions because it generates short jumps, if possible, by default. Microsoft uses `SHORT` as a flag to generate a short jump if possible.
- The assembler for x86 does not always correctly determine the size of file-scope static objects, especially when they are

declared as arrays. If you are referring to them from assembly, you should explicitly name a size, for example:

```
mov dword ptr [foo], eax
```

is preferred over:

```
mov [foo], eax
```

as the second form may generate the wrong instruction. This has been fixed in later versions of the assembler.

- The CodeWarrior assembler does not accept suffix notation hexadecimal numbers. `0x1AE3` is allowed, but `1AE3h` is not.

C/C++ Library Differences

The `extras.c` file (part of MSL for Intel platforms) defines the following functions which are equivalent to functions by the same name in Microsoft's library. These functions include:

`_chdrive()`	`_heapmin()`	`_stricmp()`
`_strnicmp()`	`_strrev()`	`_wstrrev()`
`_strdup()`	`_strupr()`	`_strdate()`
`_itoa()`	`_itow()`	`_ultoa()`
`_fullpath()`	`_alloca()`	`_makepath()`
`_searchenv()`	`_getdiskfree()`	`_getdcwd()`
`_getdrive()`	`_getdrives()`	`_strlwr()`
`_splitpath()`	`_wtoi()`	`_wcslwr()`
`_wcsupr()`	`_wcsdup()`	`_wcsicmp()`
`_wcsnicmp()`	`_wcsrev()`	`_wcsset()`
`_wcsnset()`	`_gcvt()`	

CodeWarrior also supports many of the C9X standard's new floating point functions. In some cases, there are C9X equivalents for Microsoft library functions, although their interfaces may not be the same. The MS FPU control functions `_clear87()`, `_clearfp()`,

`_control87()`, `_controlfp()`, `_status87()`, `_fpreset()`, and `_fpieee_ flt()` are supported by functions declared in the C9X `fenv.h` header file. CodeWarrior does not implement equivalents of `_chgsign()` and the Bessel functions.

Some of the functions supported are listed in Table 3.1.

Table 3.1 Microsoft functions and their ANSI/ISO equivalents

This Microsoft function…	is equivalent to this ANSI/ISO C9X function
`_finite()`	`isfinite()`
`_hypot()`	`hypot()`
`_isnan()`	`isnan()`
`_copysign()`	`copysign()`
`_nextafter()`	`nextafter()`
`_scalb()`	`scalb()`
`_fpclass()`	`fpclassify()`

4

THINK® Pascal and MPW Pascal Porting Issues

This chapter covers issues you will encounter when porting a programming project from Symantec™ THINK Pascal or Apple's MPW Pascal to the CodeWarrior IDE, including potential problems (and solutions) you might have when porting a project that run on Macintosh computers with Motorola 680x0 processors. This chapter also covers topic to help you port your application to run on PowerPC-based Macintosh computers.

This chapter has these sections:

For related information, see "Mac OS Issues" on page 14.

Pascal Compiler Differences

Notable differences between CodeWarrior Pascal compilers and other Pascal compilers for Mac OS are listed here:

Mac OS Toolbox Initialization

Applications ported from THINK Pascal to CodeWarrior Pascal for Mac OS sometimes crash immediately after they are launched. The reason: THINK Pascal has an option that automatically initializes the Mac OS Toolbox. In CodeWarrior Pascal it is up to you, the pro-

grammer, to call the proper initialization routines before using QuickDraw, Window Manager, Menu Manager, and other services.

A typical initialization routine looks like Listing 4.1.

Listing 4.1 An example of initializing the Mac OS Toolbox

```
(*
   Initialize the Toolbox before calling any
   other Mac OS routines.
*)

MaxApplZone;
MoreMasters;
InitGraf(@qd.thePort);
InitFonts;
InitWindows;
InitMenus;
TEInit;
InitDialogs(NIL);
FlushEvents(everyEvent, 0);
InitCursor;
```

For more information on Toolbox initialization, refer to *Inside Macintosh: Overview*.

Pascal Library Differences

This section describes the differences among the libraries and units that CodeWarrior Pascal and other Pascals provide for access to the Mac OS Toolbox.

Most of the topics in these sections cover problems you will encounter when porting older Pascal source code to use the Universal Pascal Interfaces, the interface files published by Apple and distributed with CodeWarrior.

The topics in this section are:

- About Universal Interfaces

About Universal Interfaces

CodeWarrior Professional always comes with the latest Universal Interfaces, the API for the Mac OS. Apple produces and regularly updates the Universal Interfaces to fix bugs, accommodate new system software features, and anticipate future system software features.

If you are porting a project that previously ran only on 68K to run on PowerPC too, the most notable feature of the Universal Interfaces is the identical access to both the 68K and PowerPC Mac OS platforms. With the Universal Interfaces, the same source code will give identical functionality when compiled for either variety of Macintosh computer.

Many of the differences between other Pascal compilers for Mac OS and CodeWarrior compilers are because of the changes the Universal Interfaces introduce. These changes are described in the following topics.

In Pascal programming for Mac OS, the Universal Interfaces also called the Universal Pascal Interfaces (UPIs).

Using Interfaces and Units

CodeWarrior Pascal requires the explicit inclusion of a USES statement in every source code file that uses the Universal Interfaces or your own units. When moving from other Pascal compilers make sure to include all relevant units.

A USES statement for a typical program might look like this:

```
USES  Types, QuickDraw, Events, Windows,
      Dialogs, Fonts, DiskInit, TextEdit,
```

```
Traps, Devices, Memory, SegLoad,
Scrap, ToolUtils, OSUtils, Menus;
```

One alternative to using this long clause in your source code is to create a unit that uses all the Universal Interfaces your program requires. Include this unit in the USES clause in each of your project's source code files. Then turn on the **Uses Propagation** option in the **Pascal Language** settings panel.

Another alternative is to use the precompiled interfaces for 68K and PowerPC, `MacOSIntf_68K` and `MacOSIntf_PPC`. Type the name of the appropriate precompiled interface in the **Prefix File** field of the Pascal Language settings panel.

NOTE: The `OSIntf` and `ToolIntf` units, which were available in MPW Pascal, are not available with the Universal Interfaces.

For more information on Universal Interfaces, see "About Universal Interfaces" on page 27.

QuickDraw Global Variables

The Universal Interfaces require you to access the QuickDraw global variables through a global record variable called `qd`. Any references made to the Quickdraw globals must use the `qd` record.

For example, the statement

```
InitGraf(@thePort);
```

becomes

```
InitGraf(@qd.thePort);
```

The QuickDraw global data that must be accessed with the `qd` qualifier are: `randSeed`, `screenBits`, `arrow`, `dkGray`, `ltGray`, `gray`, `black`, `white`, `thePort`.

For more information on Universal Interfaces, see "About Universal Interfaces" on page 27.

SANE is Obsolete

Consider SANE (Standard Apple Numerics Environment) obsolete when writing or updating software to be used on PowerPC-based Macintosh computers and on 68K-based Macintosh applications running under CFM 68K. SANE is the numerics environment Apple developed for "classic" 68K Macintosh computers.

`SANE.p` (for Pascal) is no longer part of the Universal Interfaces. Instead, use the routines provided by fp.p.

See also "68K and PowerPC Numerics" on page 29.

68K and PowerPC Numerics

For more information on porting 68K floating point operations and data types to PowerPC, refer to *Inside Macintosh: PowerPC Numerics*. Appendix A of this manual describes the differences between SANE and PowerPC numerics and offers tips on porting mathematical operations to PowerPC.

extended Data Type

The `extended` data type is not supported in PowerPC. Instead, use `double_t`, which is defined for both 68K and PowerPC. Refer to Appendix A of *Inside Macintosh: PowerPC Numerics* for more information.

For more information on Universal Interfaces, see "About Universal Interfaces" on page 27.

For information on SANE, see "SANE is Obsolete" on page 29.

Procedure Pointers, Callbacks, and UPPs

While ordinary procedure pointers work well when used only with 68K code, Universal Procedure Pointers (UPPs) are required when developing code for both 68K and PowerPC.

Specifically, UPPs allow the Mixed Mode Manager to handle 68K routine calls to PowerPC routines and PowerPC routine calls to 68K

routines. UPPs are versatile, allowing you to combine PowerPC and 68K object code in the same program without dealing with mode switches, parameter order on the stack, and other low-level issues. On the downside, converting source code to the PowerPC means explicitly creating a UPP every time you need to provide a callback routine to a Mac OS routine or any other piece of software that may either be in 68K or PowerPC object code.

Ways to create a new UPP are:

- call the `NewRoutineDescriptor` routine (in `MixedMode.p`) with the appropriate parameters
- use one of the Universal Interfaces' predeclared routines to create a specific type of UPP.

To destroy a UPP:

- call the `DisposeRoutineDescriptor` routine (in `MixedMode.p`)

For example, to create a new `ResErrUPP` in Pascal that you can pass to the Resource Manager you would call:

```
FUNCTION NewResErrProc(
  userRoutine: ResErrProcPtr): ResErrUPP;
```

passing the name of your procedure as the single parameter. `NewResErrProc` and other UPP-creating routines are implemented as glue code that is linked to your program automatically.

The Universal Interfaces provide routines to directly call universal procedure pointers. These routines are the interface to the Mixed Mode Manager.

A typical calling function might look like this:

```
PROCEDURE CallResErrProc(thErr: OSErr;
                userRoutine: ResErrUPP);
```

When you are done with this UPP, dispose of it by calling

```
DisposeRoutineDescriptor(userRoutine);
```

These calls are only needed for writing a callback to a Mac OS routine that exists in a different architecture (PowerPC calling 68K or 68K calling PowerPC).

Note that these calling functions are only needed when you write code that expects a callback parameter from an outside source. Generally this is only required if your program allows loadable submodules or patches traps that expect callback functions. Thus you would only have to use the `CallResErrProc` procedure if you install a patch to the Resource Manager that needs to use the `ResError` callback routine.

For more information on Universal Interfaces, see "About Universal Interfaces" on page 27.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

L

M

T

U

V

W

X

Y

CodeWarrior

Porting Reference

Credits

writing lead: Marc Paquette

other writers: Carl B. Constantine, Gavriel State, L. Frank Turovich

engineering: the entire Metrowerks R&D staff

frontline warriors: Steve Chernicoff, Ben Combee, Joe Hayden, Rick Grehan, Steve Jovanovic, Jun-Kiat Lam, Ron Liechty, John Roseborough, Lucien Stavenhagen, Jim Trudeau, L. Frank Turovich, and CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows