

Freescale Semiconductor, Inc.

**CodeWarrior™**  
**Development Tools**  
**Profiler User Guide**

**metrowerks**

For More Information: [www.freescale.com](http://www.freescale.com)

# Freescale Semiconductor, Inc.

Metrowerks, the Metrowerks insignia, and CodeWarrior are registered trademarks of Metrowerks Corp. in the US and/or other countries. All other trade names, trademarks and registered trademarks are the property of their respective owners.

© Copyright. 2002. Metrowerks Corp. ALL RIGHTS RESERVED.

Metrowerks reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Metrowerks does not assume any liability arising out of the application or use of any product described herein. Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft or automobile navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.

Documentation stored on electronic media may be printed for personal use only. Except for the forgoing, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

## How to Contact Metrowerks:

<b>Corporate Headquarters</b>	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
<b>World Wide Web</b>	<a href="http://www.metrowerks.com">http://www.metrowerks.com</a>
<b>Sales</b>	United States Voice: 800-377-5416 United States Fax: 512-996-4910 International Voice: +1-512-996-5300 Email: <a href="mailto:sales@metrowerks.com">sales@metrowerks.com</a>
<b>Technical Support</b>	United States Voice: 800-377-5416 International Voice: +1-512-996-5300 Email: <a href="mailto:support@metrowerks.com">support@metrowerks.com</a>

**For More Information: [www.freescale.com](http://www.freescale.com)**

# Freescale Semiconductor, Inc. Table of Contents

---

<b>1 Introduction</b>	<b>7</b>
Read the Release Notes! . . . . .	7
What's New in This Release . . . . .	7
CodeWarrior and Its Documentation . . . . .	8
What's in This Manual . . . . .	8
Where to Go from Here . . . . .	9
<b>2 Getting Started</b>	<b>13</b>
System Requirements . . . . .	13
Installing Profiler . . . . .	14
Background on Profiling Code . . . . .	14
What Is a Profiler? . . . . .	15
Types of Profilers . . . . .	15
A Profiling Strategy . . . . .	17
Profiling Code . . . . .	18
<b>3 Using Profiler</b>	<b>19</b>
What It Does . . . . .	19
How It Works . . . . .	20
Profiling Made Easy . . . . .	21
<b>4 Viewing Results</b>	<b>27</b>
What It Does . . . . .	27
How It Works . . . . .	27
MW Profiler Window . . . . .	28
Window Views . . . . .	32
Finding Performance Problems . . . . .	36
<b>5 Profiling Mac OS Code</b>	<b>39</b>
Profiler Libraries and Interface Files. . . . .	39
Profiling Special Cases . . . . .	42
Profiling Code with #pragma Statements . . . . .	43
Initializing Profiler with ProfilerInit() . . . . .	43
Terminating Profiler with ProfilerDump() . . . . .	46
Profiling Threads . . . . .	47
Viewing Threads in MW Profiler . . . . .	48

# Freescale Semiconductor, Inc.

## Table of Contents

---

Profiling Shared Libraries . . . . .	49
Profiling Code Resources . . . . .	49
Profiling C++ Static Constructors . . . . .	50
Profiling MacApp . . . . .	51
Profiling Asynchronous Routines . . . . .	51
Profiling Abnormally Terminated Functions . . . . .	52
Using the PowerPC PEF Linker . . . . .	53
Debugging Profiled Code . . . . .	58
Profiler Limitations . . . . .	59
<b>6 Profiling in PowerPlant . . . . .</b>	<b>61</b>
Introduction to Profiling in PowerPlant . . . . .	61
Profiling Strategy . . . . .	61
Rules of Thumb . . . . .	62
Profiling Class . . . . .	63
StProfileSection . . . . .	63
~StProfileSection . . . . .	64
Profiling Your Code . . . . .	65
Code Exercise for Profiling . . . . .	66
Profiling a Single Routine . . . . .	66
Profiling an Application . . . . .	71
<b>7 Troubleshooting . . . . .</b>	<b>73</b>
Profile Times Vary Between Runs . . . . .	73
Problems while Profiling Inline Functions . . . . .	74
Profiling Library Could not be Found . . . . .	75
Profiler and Virtual Memory . . . . .	76
Names are Garbled when Viewing a 68K Profile Dump . . . . .	76
<b>8 Profiler Reference . . . . .</b>	<b>77</b>
Compiler Directives . . . . .	77
Testing for the Profiler . . . . .	78
Memory Usage . . . . .	78
Time and Timebases . . . . .	79
Profiler Menu Reference . . . . .	80
About MW Profiler . . . . .	81
File Menu . . . . .	81
Edit Menu . . . . .	82

---

View Menu . . . . .	82
Windows Menu . . . . .	84
Profiler Function Reference . . . . .	84
Profiler API . . . . .	85
<b>Index</b>	<b>91</b>



# Introduction

---

Welcome to the CodeWarrior Profiler User Guide. Profiler is a code analysis tool designed to help you make your code more efficient.

The introduction includes the following sections:

- Read the Release Notes!—where to go for critical, last-second details
- What's New in This Release—new features since the previous release
- CodeWarrior and Its Documentation—a general description of the CodeWarrior architecture and documentation
- What's in This Manual—a description of the contents of this manual
- Where to Go from Here—recommendations for further reading

## Read the Release Notes!

Before you use CodeWarrior Profiler, you should read the release notes. They contain important last-minute information about new features, bug fixes, and incompatibilities that *may not be included in the documentation*.

The release notes directory is always included as part of a standard CodeWarrior installation. The release notes directory is also located at the top level of the CodeWarrior CD.

## What's New in This Release

MW Profiler is Carbon compliant and thus requires the CarbonLib from Apple to run. A version of CarbonLib is available on the CodeWarrior CD.

## CodeWarrior and Its Documentation

CodeWarrior is a multi-host, multi-language, multi-target development environment. What does that mean?

**Multiple hosts** CodeWarrior runs on several different operating systems, including Windows and Mac OS. The features, human interface, and operation of CodeWarrior are very similar on all hosts.

**Multiple languages** You can use CodeWarrior to program in several languages, including C/C++, and Java. Third-party compilers provide support for other languages, such as Fortran. Which languages are available to you depend upon the target for which you are developing software.

**Multiple targets** You can use CodeWarrior to write software for several different chips or operating systems. CodeWarrior products support programming for a variety of embedded processors and real-time operating systems, the Java Virtual Machine, and desktop operating systems such as Windows, Mac OS.

Most features of CodeWarrior apply regardless of your preferred host, language, or target. General features of CodeWarrior are described in the *IDE User Guide*.

For a complete understanding of CodeWarrior, you must refer to both the general documentation and the documentation that is specific to your particular target.

While the profiling process is general, each target has its own unique features. This manual describes both the general profiling process and the unique features involved in profiling code for specific targets.

## What's in This Manual

The CodeWarrior Profiler is actually a system consisting of three components: the profiler libraries, the calls you add to your code, and the MW Profiler application you use to view results. The Profiler system works with C, C++, and Pascal code.



This manual often refers to the CodeWarrior profiler system as “CodeWarrior profiler,” or simply as “the profiler.” This manual will specifically refer to the MW Profiler application as “MW Profiler.”

Table 1.1 lists every chapter in this manual, and describes the information contained in each. See “CodeWarrior and Its Documentation” on page 8 for a discussion of how these chapters relate to other CodeWarrior documentation.

**Table 1.1 Contents of chapters**

<b>Chapter</b>	<b>Description</b>
Introduction	this chapter
Getting Started	system requirements, installation instructions, and background information
Using Profiler	a “how to” guide to profiling your code
Viewing Results	a user’s guide to displaying profiling results with MW Profiler
Profiling Mac OS Code	information about profiling Mac OS code, including a tutorial
Troubleshooting	common problems and their answers
Profiler Reference	a reference to the libraries, compiler directives, and function calls you use to control the CodeWarrior profiler at compile time

## Where to Go from Here

All the manuals mentioned here are available as part of the CodeWarrior documentation included with your product.

# Freescale Semiconductor, Inc.

## If you're new to CodeWarrior Profiler:

- If you are unfamiliar with what a profiler is, read "What Is a Profiler?." This section discusses different approaches to profiling code, and why profiling is so useful.
- Read "Using Profiler" and "Viewing Results." These chapters introduce you to the three components of the CodeWarrior profiler system: the profiler libraries, the calls you add to your code, and the MW Profiler application you use to view results.

## If you're experienced with CodeWarrior Profiler:

- For details about profiling Mac OS code, read "Profiling Mac OS Code."
- If you are experiencing problems using any part of the CodeWarrior profiler system, read "Troubleshooting."
- For technical details on the CodeWarrior profiler system, including the libraries, API, and other issues, see "Profiler Reference."
- For reference on menu items in the MW Profiler application, use "Profiler Menu Reference" as your starting point.

## For everyone:

- For general information about the CodeWarrior IDE and debugger, see the *IDE User Guide*.
- For information specific to the C/C++ front-end compiler, see the *C Compilers Reference*.
- For information on Metrowerks' standard C/C++ libraries, see the *MSL C Reference* and the *MSL C++ Reference*.

## For general information on profiling Mac OS code:

- Read "Optimizing PowerPC Programs," *Apple Directions*, Nov. 95, p. 17.
- Read "Balance of Power," *develop*, Issues 18 (p. 55) and 19 (p. 17)
- Read "PowerPC Compatibility and Performance Issues," *Tech Note PT38*.
- Read "Taking Extreme Advantage of PowerPC," *MacTech Magazine*, Eric Traut, April 95
- Read "Preparing Your Code for Future PowerPC Processors," *Apple Directions*, May 95, p. 14.

There is a complete index at the back of this manual.

# Freescale Semiconductor, Inc.

**Introduction**  
*Where to Go from Here*

---

# Getting Started

---

This chapter gives you the information you need to get CodeWarrior Profiler installed and running. For new users, this chapter provides a brief overview of profiling code.

This chapter includes the following topics:

- System Requirements — hardware and software requirements
- Installing Profiler— how to install Profiler on your system
- Background on Profiling Code— a brief description of profilers and how to implement them in your code

## System Requirements

This section describes what kind of system you'll need in order to use CodeWarrior Profiler.

Profiler is part of a complete package that also includes the CodeWarrior IDE, MW Profiler, and the profiler libraries. You need all these elements to use the Profiler system effectively.

- A Macintosh-compatible computer with a PowerPC 601 or better processor.
- Mac OS System 8.1 or later with CarbonLib 1.02 or later installed.
- A minimum of 32 MB RAM to use the CodeWarrior IDE along with MW Profiler.
- A CD-ROM drive to install CodeWarrior software, documentation, and examples

## Installing Profiler

Your first step toward developing software for your target is to install the CodeWarrior tools. The tools include a variety of components such as the IDE, debugger, plug-in compilers and linkers, standard libraries, runtime libraries and headers, and all necessary documentation.

The CodeWarrior Installer automatically installs all necessary components. It is *strongly* recommended that you use the CodeWarrior Installer to ensure that you have all the required files. If you have any questions regarding the installer, read the instructions built into the CodeWarrior Installer itself.

To start the installation process, do the following:

On the Macintosh desktop, double-click the icon for the CodeWarrior CD. Then double-click the icon for the CodeWarrior Installer, located at the root level of the CD.

## Background on Profiling Code

This section provides you with general information about what a profiler is, different kinds of profilers, and a typical strategy you would follow to measure program performance. Along the way you'll see how the CodeWarrior profiler handles the advantages and disadvantages of profiling code.

Topics discussed are:

- What Is a Profiler?—a brief description of profilers and what they do
- Types of Profilers—different kinds of profilers, their strengths and weaknesses
- A Profiling Strategy—an outline you should follow when profiling your own code
- Profiling Code—three steps to follow when profiling code

## What Is a Profiler?

Speed and performance are important issues in most software projects. In most cases, if your code doesn't work quickly, it doesn't work well.

Programmers have regularly observed that 10% of their code does 90% of the work. Reworking code to make it more efficient is a non-trivial task. You should concentrate on improving that core 10% of your code first, and improve the infrequently-used code later, if at all.

Wouldn't it be really cool if you could determine precisely where your code spent its time? That's what a profiler does for you—it gives you clues. More than clues, the CodeWarrior profiler gives you hard and reliable data.

A good profiler analyzes the amount of time your code spends performing various tasks. Armed with this information, you can apply your efforts to improving the efficiency of core routines.

A profiler can also help you detect bottlenecks—routines your data passes through to get to other places—and routines that are just inordinately slow. Identifying these problems is the first step to solving them.

## Types of Profilers

The simplest profilers count how many times a routine is called. They do not report any information about which routines are called by other routines, or the amount of time spent inside the various routines being profiled.

Clearly a good profile of the runtime performance of code requires more information than a raw count. More advanced profilers perform statistical sampling of the runtime environment. These profilers are called passive or sampling profilers.

A passive profiler divides the program being profiled into evenly-sized "buckets" in memory. It then samples the processor's program counter at regular intervals to determine which bucket the counter is in.

The main advantage of a passive profiler is that it requires no modification to the program under observation. You just run the profiler and tell it what program to observe. Also, passive profilers distribute the overhead that they incur evenly over time, allowing the post-processing steps to ignore it. On the other hand, they cannot sample too frequently or the sampling interrupt will overwhelm the program being sampled.

Passive profilers have a significant disadvantage. Although useful, bucket boundaries do not line up with routine boundaries in the program. This makes it difficult if not impossible to determine which *routines* are heavily used. As a result, passive profilers generate a relatively low-resolution image of what's happening in the program while it runs.

In addition, because they rely on a statistical sampling technique, the program must run for a long enough period to collect a valid sample. As a result, they do not have good repeatability—that is, the results you get from different runs may vary unless the sampling period is long.

The most advanced and accurate profilers are called active profilers. The CodeWarrior profiler is an active profiler.

An active profiler tracks the precise amount of time a program spends in each individual routine, measured directly from the system clock.

To perform this magic, an active profiler requires that you modify the code of the program to be observed. An active profiler gains control at every routine entry and exit. There must be a call to the profiler at the beginning of each profiled routine. The profiler can then track how much time is spent in the routine.

This approach has significant advantages over a passive profiler. An active profiler can report high-resolution results about exactly what your program is doing. An active profiler also tracks the dynamic call tree of a program. This information can be very useful for determining the true cost of calling a routine. The true cost of a routine call is not only the time spent in the routine, it is also the time spent in its children—the subsidiary routines it calls, the routines they call, and so on to whatever depth is necessary.



Because it uses measurements and not statistical sampling, an active profiler is much more accurate and repeatable than a passive profiler.

The requirement that you must modify the actual source code might seem like a significant disadvantage. With the CodeWarrior profiler, this disadvantage is minimal. Activating the profiler for an entire program—or for a range of routines within a program—is simple. The compiler does most of the work, inserting the necessary calls to the profiler itself. You do have to recompile the project when you turn on profiling.

Active profilers distribute their overhead by routine call frequency. This makes it harder to compensate for the overhead during post processing of the output. The CodeWarrior profiler tracks the overhead automatically, compensates for it, and also reports the overhead to you. You don't have to worry about overhead at all.

Finally, active profilers generate a large amount of raw information. This can lead to confusion and difficulty interpreting the results. The MW Profiler application that is part of the CodeWarrior profiler system handles these difficulties with aplomb. You can view and sort the data in whatever way best suits your needs.

## A Profiling Strategy

You use a profiler to measure the runtime performance of your code. What is usually important is how your code's performance measures up to some standard. When approaching the problem of measuring performance, you might want to take these three steps:

- 1. Establish your standards.**

For example, you might decide that you want the program to load in less than ten seconds, or check the spelling of a five-page document that contains no misspellings in 15 seconds. Also decide on the platform you will use for testing, since processor speeds vary.

- 2. Determine how to measure time.**

Your measurement device may be no more complicated than a stopwatch, or you may need to add some simple code to count ticks. At this phase you want to test the code in as close to its finished

form as possible, so measure time in a way that is accurate enough to suit your needs, and that has the lowest impact on your code's natural performance. You do not want to run a full-blown profile here, because profiling can add significant overhead, thus slowing down your code's raw performance.

**3. Run the tests and measure results.**

If you meet your performance goals, your job is done. If your code does not meet your goals, then it's time to profile your code.

## Profiling Code

To profile your code, you do three things:

**1. Run a profiler on the area of the code you want tested.**

This might be a single routine, a group of routines that perform a task, or even the entire application. What you profile depends upon what you are testing.

**2. Analyze the data collected by the profiler and improve your code.**

You study the results of your profiling and look for problems and room for improvement.

The profiling process is iterative. You repeat these two steps until you achieve the performance gain you need to meet your goals.

The rest of this manual discusses how to perform these two steps—profile your code and analyze the results—using the CodeWarrior profiler system.

**3. Retest your code to verify results**

When you are satisfied that you have reached your goals, you have one more step to perform. You should run your original tests—without the profiler of course—to verify that your code in its natural state meets your performance goals.

The CodeWarrior profiler will help you meet those goals quickly and easily.

# Using Profiler

---

The CodeWarrior profiler lets you analyze how processor time is distributed during your program's execution. With this information, you can determine where to concentrate your efforts to optimize your code most effectively.

This chapter discusses the following principal topics:

- What It Does—an overview of the principle features of the profiler
- How It Works—basic information on the elements of the profiler and about how to use the profiler in your own code
- Profiling Made Easy—a step-by-step guide to using the profiler

## What It Does

The CodeWarrior profiler is a state-of-the-art, user-friendly, analytical tool that can profile C or C++ code.

For every project, from the simplest to the most complex, the profiler offers many useful features that help you analyze your code. You can:

- turn the profiler on and off at compile time
- profile any routine, group of routines, or an entire project
- track time spent in any routine
- track time spent in a routine and the routines it calls—its children
- track execution paths and times in a dynamic call tree
- collect detailed or summary data in a profile
- use precision time resolutions for accurate profiling
- track the stack space used by each routine

## How It Works

The CodeWarrior profiler is an active profiler. The profiling system consists of three main components:

- a statically-linked code library of compiled code containing the profiler
- an Application Programming Interface (API) to control the profiler
- the MW Profiler application to view and analyze the profile results

Details of the API are discussed in “Profiler API.” The MW Profiler application is discussed in “Viewing Results.”

The rest of this chapter will discuss the general profiling process. Subsequent chapters describe how to carry out the profiling process for your particular target.

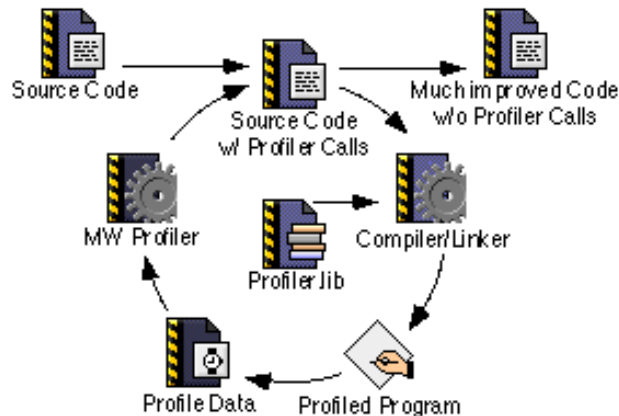
To use the profiler, you do these things:

- Include the correct profiler library and files in your CodeWarrior project
- Modify your source code to make use of the profiler API
- Use the API to initialize the profiler, to dump the results into a file, and to exit the profiler
- Use the MW Profiler application to view the results

You can profile an entire program if you wish or, adding compiler directives to your code, you can profile any individual section of your program.

Figure 3.1 illustrates the flow of events that occurs when you profile code.

Figure 3.1 The profiling process



You modify the original source code slightly to initialize the profiler, dump results, and exit the profiler when through. You may modify the source code more extensively if you wish to profile individual portions of your code.

Then the compiler and linker—using a profiler library—generate a new version of your program, ready for profiling. While it runs, the profiler generates data. Your program will run a little more slowly because of the profiler overhead (sometimes a *lot* more slowly), but that’s taken into account in the final results. When complete, you use the MW Profiler application to analyze the data and determine what changes are appropriate to improve performance. You can repeat the process as often as desired until you have turned your code into a fast, efficient, well-oiled machine.

See also “Profiler API” and “Viewing Results.”

## Profiling Made Easy

This section takes you step by step through the general process of profiling an application.

To profile an application, you do the following:

1. Add a profiler library to the project

2. Turn on profiling
3. Include the profiler API interface
4. Initialize the profiler
5. Dump the profile results
6. Exit the profiler

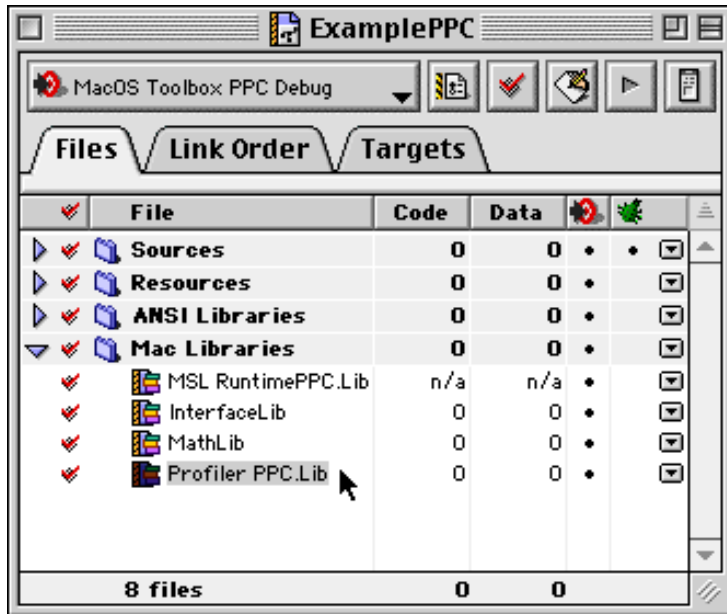
In the steps that follow, we'll detail precisely what to do in both C/C++ and Pascal. These steps may seem a little complicated. Don't be alarmed. Using the CodeWarrior profiler is actually easier than reading about how to do it.

## 1. Add a profiler library to the project

The code that performs the profiler magic has been compiled into libraries. The precise library that you add to your code depends on the target for which you're profiling code and on the kind of code you're developing. For more information about adding the appropriate libraries for your particular target, read "Profiling Mac OS Code."

For example, Figure 3.2 shows how to add the appropriate library for your code model when profiling code for a Mac OS PowerPC target.

Figure 3.2 The PowerPC profiler library in a project



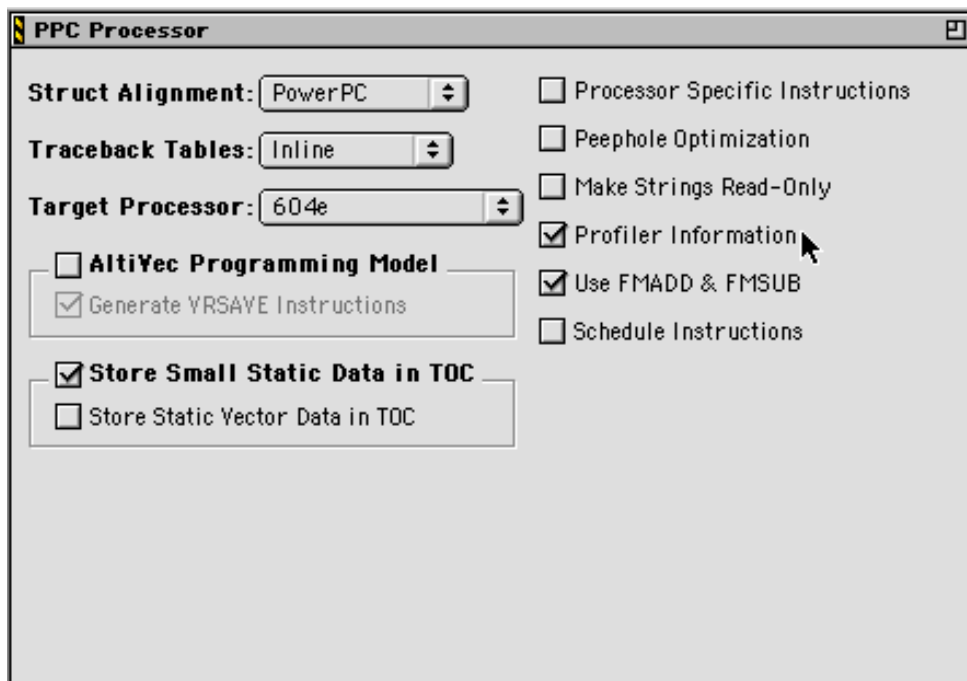
2. Turn on profiling

You can use the following methods to turn profiling on or off:

**Project-level profiling**

To turn on profiling for an entire project, use the project settings. In the Project Settings dialog, choose the processor you are generating code for (68K Processor or PPC Processor) under the Code Generation option. Click the **Generate Profiler Information** checkbox as shown in Figure 3.3. With profiling on, the compiler generates all the code necessary so that every routine calls the profiler.

**Figure 3.3 Processor Preferences options for PowerPC**



### ***Routine-level profiling***

To profile certain routines (rather than the entire project), use the appropriate profiler API calls for your target to initialize the profiler, set up profiling, and immediately turn profiling off. You can then manually turn profiling on and off by placing profiler calls around the routine or routines you want to profile. For example, you could modify Mac OS code to look like Listing 3.1.

#### **Listing 3.1 profiling a routine**

```
void main()
{
    ...
    err = ProfilerInit(...);

    if (err == noErr)
    {
```



```
ProfilerSetStatus(FALSE); // turn off profiling until needed.
// more code....

// now you reach routine you want to profile
ProfilerSetStatus(TRUE); // turn on profiling
foobar(); // this routine is profiled and shows up in viewer
ProfilerSetStatus(FALSE); // turn profiling off again

// more code....
ProfilerTerm();
}
}
```

Assuming that profiling is on for an entire project, you can turn off profiling at any time. First, use an appropriate call to turn off profiling. Then use another call to turn it on. Turn it on just before calling the routine or routines you are interested in. Turn it off when those routines return. It's really that easy.

Alternatively, you can use `#pragma` statements in C/C++. These aren't as useful as using profiler API calls. For example, suppose you have two routines—`foo()` and `bar()`—that each call a third utility routine, `barsoom()`. If you use compiler directives to turn on profiling for `foo()` and `barsoom()`, the result you get will include the time for `barsoom()` when called from `bar()` as well.

### 3. Include the profiler API interface

To use the profiler, you add at least three profiler-related calls to your code. These calls are detailed in the next three steps. The process varies slightly for the different languages and targets.

Source files that make calls to the profiler API must include the appropriate header file for your target. For example, to profile an entire Mac OS application, you would add this line of code to the source file that includes your `main()` function:

```
#include <profiler.h>
```

---

**TIP** You don't have to include the header file in every file that contains a profiled function, only in those that actually make direct profiler API calls.

---

#### 4. Initialize the profiler

At the beginning of your code, you call the appropriate function for your target. See “Profiler API” to find out the precise function name that you’ll need for your specific target.

#### 5. Dump the profile results

Obviously, if you profile code you want to see the results. The profiler dumps the results to a data file. The data is in a proprietary format understood by the MW Profiler viewing application.

---

**TIP** Although the original profiler data file is in a proprietary format, you can use the **Save As...** command in the MW Profiler **File** menu to save the data in a tab-delimited text file.

---

#### 6. Exit the profiler

When you are all through with the profiler, before exiting the program you *must* terminate the profiler by calling the correct profiler API function. If you initialize the profiler and then exit the program without terminating the profiler, timers may be left running that could crash the machine.

The call to terminate the profiler stops the profiler and deallocates memory. It does not dump any information. Any collected data that has not been dumped is lost when you call the function to terminate the profiler.

Having performed these quick steps, you simply compile your program and run it. When you quit, the results will be waiting for your analysis. MW Profiler, the viewing application, is introduced in “Viewing Results.”

In summary, the process of using the CodeWarrior profiler is quite easy. You add the requisite library, turn on profiling, include the header file, initialize the profiler, dump the results, and exit. It is a remarkably painless and simple process that quickly gets you all the data you need to perform a professional-level analysis of your application’s runtime behavior.

# Viewing Results

---

This chapter discusses MW Profiler—the application you use to view the data created by the CodeWarrior profiler.

In this chapter you will look at:

- What It Does—the principle features of MW Profiler
- How It Works—the MW Profiler interface and how you can view data
- Finding Performance Problems—use the MW Profiler to locate problems

## What It Does

The MW Profiler viewer application displays profiler output for you to analyze the results of your program's execution. MW Profiler reads the profiler dump files created by the calls in your code and displays the data in a form that you can use. Using the data display you can:

- sort data by any of several relevant criteria such as name, time in routine, percent of time in routine, and so forth
- open multiple profiles simultaneously to compare different versions of the profiled code
- identify trouble spots in the code
- view summary, detailed, or object-based data

The MW Profiler application is very easy to use.

## How It Works

You open profile data files exactly as you open files in any application. You can use the **Open...** command from the MW

Profiler **File** menu or drop the data file's icon on the MW Profiler icon. Whatever approach you take, when you open a file a window appears.

## MW Profiler Window

MW Profiler allows you to view several elements of the profile data simultaneously, as shown in Figure 4.1.

**Figure 4.1 Profiler window**

Collection Method	Timebase	Time/Date created	Overhead (ms)			
ExamplePPC.prof						
Method: Detailed Timebase: PowerPC Saved at: 11:32:18 5/23/97 Overhead: 0.504						
Function Name	Count	Only	%	+Children	%	Average
test	1	0.032	0.0	2492.671	100.0	0.032
test2	15	1245.542	50.0	2492.639	100.0	83.036
test3	15	1246.902	50.0	1246.981	50.0	83.127
test4	30	0.195	0.0	0.195	0.0	0.007

### Profiler window header items (Mac OS)

MW Profiler allows you to view summary data, detailed data, or object-based data. All three ways of viewing data use the same window shown in Figure 4.1. The display of the contents changes slightly for the different views.

See also "Window Views" on page 32.

At the top of the window is a line with information about the data on display. The figure shows four items, but six are possible. What they are and what they represent is listed in Table 4.1.

**Table 4.1 Profile window header items**

Item	Contents
Method	the collection method, detailed or summary
Timebase	the time base that was used to collect the profile information
Saved at	the date and time the profile data was created
Overhead	time (in milliseconds) used by the profiler
OverflowStack	the number of routines that were called when the profiler's internal stack was overflowing
OverflowFunctions	the number of routines called that did not fit in the profiler's routine table

The last two items—OverflowStack and OverflowFunctions—only appear if the parameters you passed to `ProfilerInit()` did not create a buffer large enough to hold all the collected data.

The OverflowStack number includes duplicate routine calls. If it is small, it is the amount by which you should increase the `stackDepth` parameter in the call to `ProfilerInit()`. If it is large, increase the value of the `stackDepth` parameter by some increment and try again.

The OverflowFunctions number also includes duplicate routine calls. It is likely to be much larger than the size by which you need to increase the `numFunctions` parameter in the call to `ProfilerInit()`. It might even be in the thousands.

If either overflow occurs, just increase the value of the respective parameter in the `ProfilerInit()` call so the profiler allocates a larger data buffer.

See also “**ProfilerInit()**.”

### Profiler window data columns

Below the header information is a series of columns containing data from the profile. All times are displayed according to the resolution of the timer that you use to profile data. The results in the window are only as precise as the timer used.

The times shown in the data columns are displayed in milliseconds. A millisecond (ms) is one thousandth of a second. Each time datum is reported to three decimal places. This gives you microsecond results. A microsecond ( $\mu$ s) is one thousandth of a millisecond, or one millionth of a second. However, some time bases (most notably `ticksTimeBase`) are less precise. Ticks increment 60 times a second. See "Time and Timebases."

## Resizing data columns

Figure 4.1 does not show all the data columns. You can scroll to see the columns beyond the right edge of the window. You can also resize the columns to get more information in the window.

Move the cursor over one of the column lines. The cursor changes to a directional arrow, as shown in Figure 4.2. Click and drag the column line to where you want it.

**Figure 4.2** Resize columns

Method: Detailed Timebase: PowerPC Saved at: 11:32:18 5/23/97 Overhead: 0.504						
Function Name	Count	Only	%	+Children	%	Average
test	1	0.032	0.0	2492.671	100.0	0.032
test2	15	1245.542	50.0	2492.639	100.0	83.036
test3	15	1246.902	50.0	1246.981	50.0	83.127
test4	30	0.195	0.0	0.195	0.0	0.007

Table 4.2 lists each of the columns in the profiler window (from left to right) and the information that column contains.

Table 4.2 Profile window data columns

Column	Contents
Function name	the routine name (if it is a C++ function name, the profiler unmangles it)
Count	the number of times this routine was called
Only	time spent in the routine itself without counting any time in routines called by this routine
%	percent of total time for the Only column
+Children	time spent in this routine and all the routines it calls.
%	percent of total time for the +Children column
Average	the average time for each routine invocation. (the Only time divided by the number of times the routine was called)
Maximum	the longest time for an invocation of the routine
Minimum	the shortest time for an invocation of the routine
Stack Space	(Mac OS) the largest size (in bytes) of the stack when the routine is called (for 68K code the actual size may be slightly larger)

**NOTE** If you're profiling Mac OS code, you should know that there is a slight difference between the 68K and the PowerPC in the stack tracking code. On a 68K machine the profiler is called before a routine sets up its stack. Therefore, the stack values that a 68K profile reports *do not include the called routine's stack*. On a PowerPC, the profiler is called after a routine sets up its stack. Therefore, the stack values on a PowerPC profile do include the called routine's stack.

### Sorting data

You can view the data sorted by the value in any column. Names are listed alphabetically from top to bottom, numbers are listed from greater to lesser.

To change the sort order, either click the column title, or choose a menu item from the **View** menu. The heading becomes underlined and data is sorted by the value in that column.

## Exporting data

If you wish to export information from the results, use the **Save As...** command in the **File** menu. This saves all the data in a tab-delimited text file. The text file is organized the same way that the window is organized.

## Multiple windows

You can open any number of different profile windows simultaneously, limited only by the MW Profiler's memory allotment. This allows you to compare the results of different runs easily.

## Window Views

In the MW Profiler **View** menu you may choose to view the data in one of three ways: summary, detailed, or object. Not all possibilities are available for all profiles.

### Summary View

The summary view displays a complete, non-hierarchical, flat list of each routine profiled. No matter what calling path was used to reach a routine, the profiler combines all the data for the same routine and displays it on a single line. Figure 4.3 shows a summary view.



Figure 4.3 Summary view

Function Name	Count	Only	%	+Children	%	Average
test	1	0.032	0.0	2492.671	100.0	0.032
test2	15	1245.542	50.0	2492.639	100.0	83.036
test3	15	1246.902	50.0	1246.981	50.0	83.127
test4	30	0.195	0.0	0.195	0.0	0.007

The summary view is particularly useful for comparing routines to see which take the longest time to execute. The summary view is also useful for finding a performance problem with a small routine that is called from many different places in the program. This view helps you look for the routines that make heavy demands in time or raw number of calls.

A summary view window can be displayed for any profile.

**Detailed View**

The detailed view displays routines according to the dynamic call tree as shown in Figure 4.4.

Figure 4.4 Detailed view

Function Name	Count	Only	%	+Children	%	Average
test	1	0.032	0.0	2492.671	100.0	0.032
test2	15	1245.542	50.0	2492.639	100.0	83.036
test3	15	1246.902	50.0	1246.981	50.0	83.127
test4	15	0.078	0.0	0.078	0.0	0.005
test4	15	0.117	0.0	0.117	0.0	0.008

Routines that are called by a given routine are shown indented under that routine. This means that a routine may appear more than once in the profile if it called from different routines. This makes it difficult to tell how much total time was spent in a routine. However, you can use the summary view for that purpose.

The detailed view is useful for detecting design problems in code. MW Profiler in detailed view lets you see what routines are called how often from what other routines. Armed with knowledge of your code's underlying design, you may discover flow-control problems.

For example, you can use detailed view to discover routines that are called from only one place in your code. You might decide to fold that routine's code into the caller, thereby eliminating the routine call overhead entirely. If it turns out that the little routine is called thousands of times, you can gain a significant performance boost.

You can use the **Expand All** or **Collapse All** commands in the **View** menu to open or close the entire hierarchy at once.

In detailed view, sorting is limited to routines at the same level in the hierarchy. For example, if you sort by routine name, the routines at the top of the hierarchy will be sorted alphabetically. For each of those first-level routines, its second-level routines will be sorted alphabetically underneath it, and so on.

The detailed view requires that `collectDetailed` be passed to `ProfilerInit()` when collecting the profile. If `collectSummary` is used, you cannot display the data in detailed view.

## Object View

The object view displays summary information sorted by class. Beneath each class the methods are listed. This is a two-level hierarchy. You can open and close a class to show or hide its methods, just like you can in the detailed view.

When sorting in object view, functions stay with their class, just like subsidiary functions in detailed view stay in their hierarchical position. Figure 4.5 shows the methods sorted by count.

Object view allows you to study the performance impact of substituting one implementation of a class for another. You can run profiles on the two implementations, and view the behavior of the different objects side by side. You can do the same with the summary view on a routine-by-routine basis, but the object view gives you a more natural way of accessing object-based data. It also allows you to gather all the object methods together and view them simultaneously, revealing the effect of interactions between the object's methods.

**Figure 4.5 Object view**

The screenshot shows a window titled "Object Profile" with a menu bar and a toolbar. Below the title bar, it displays "Method: Detailed Timebase: PowerPC Saved at: 10:20:04 9/9/97 Overhead: 0.258". The main content is a table with the following data:

Function Name	Count	Only	%	+Children	%	Average
▶ LApplication::	2	0.025	1.2	2.030	100.0	0.013
▶ LArray::	13	0.018	0.9	0.027	1.3	0.001
▶ LArrayIterator::	17	0.039	1.9	0.087	4.3	0.002
▶ LCommander::	4	0.033	1.6	0.090	4.4	0.008
▶ UAppleEventsMgr::	2	1.915	94.3	1.915	94.3	0.958

Object view will display "N/A" (Not Available) in the +Children column for classes in a collectSummary profile. This is because the detail information is missing from the file.

The object view requires that the profile contain at least one mangled C++ name. If there is none, you cannot use object view.

## Finding Performance Problems

As you work with MW Profiler, you will see that the information provided quickly guides you to problem areas.

To look for time hogs, sort the view by either the Only column or the +Children column. Then examine routines that appear near the top of the list. These are the routines that swallow the greatest percentage of your code's time. Any improvement in these routines will be greatly magnified in your code's final performance.

You may also want to sort based on the number of times a routine is called. The time you save in a heavily-used routine is saved each time it is called.

If stack size is a concern in your code, you can sort based on the Stack Space column. This lets you see the largest size the stack

reached during the profile. Remember that in Mac OS 68K code the stack for the called routine is not included in the Stack Space data.



# Profiling Mac OS Code

---

This reference section discusses profiling Mac OS code and associated special cases.

The sections in this chapter are:

- Profiler Libraries and Interface Files—the libraries and interface files that you add to your code in order to use the profiler
- Profiling Special Cases—special cases to consider when profiling code
- Profiler Limitations—important information for profiling 68K code

## Profiler Libraries and Interface Files

You can find all of the profiler libraries and interface files in the Profiler folder in the Metrowerks Utilities folder. The profiling code that actually keeps track of the time spent in a routine exists in a series of libraries. Depending upon the nature of your project and the platform for which you are writing code, you link in one or another of these libraries as appropriate.

**See also** “Add a profiler library to the project” and “Profiling Special Cases” on page 42.

The profiler libraries require the Macintosh operating system. You must have the appropriate Mac OS library added to your project as listed in Table 5.1. Your project may need other libraries as well.

**Table 5.1** Mac OS libraries

Language	68K	PowerPC
C/C++	MacOS.lib	InterfaceLib

# Freescale Semiconductor, Inc.

## Profiling Mac OS Code

### Profiler Libraries and Interface Files

Table 5.2 lists the libraries you use for profiling. Both Pascal and C use the same libraries. For 68K code, use the “Small” library for projects using the near code model, and “Large” for the far code and smart code models.

**Table 5.2 Profiler libraries**

Library	Processor	When To Use
Profiler 68k.Lib	68K (A5 globals)	near/small model applications
Profiler Fa(68k).Lib	68K (A5 globals)	far/large/smart model applications
Profiler 68k.A4.Lib	68K (A4 globals)	near/small model code resources
Profiler Fa(68k.A4).Lib	68K (A4 globals)	far/large/smart model code resources
Profiler CFM68k.Lib	68K (A5 globals)	code fragments and shared libraries
Profiler PPC.Lib	PowerPC	PowerPC applications and shared libraries
Profiler PPC.MP.Lib	PowerPC	PowerPC multiprocessor applications and shared libraries
ProfilerMPLib	PowerPC	PowerPC multiprocessor code fragments and shared libraries.
ProfilerLib	CFM 68K & PowerPC	code fragments and shared libraries
Profiler Carbon.Lib	PowerPC (Carbon)	PowerPC applications and shared libraries for use on Mac OS X or Mac OS 8.1 and above with the Carbon libraries
ProfilerCarbonLib	PowerPC (Carbon)	PowerPC code fragments and shared libraries for use on Mac OS X or Mac OS 8.1 and above with the Carbon libraries



**TIP** Use `ProfilerLib`, `ProfilerMPLib`, and `ProfilerCarbonLib` when you want to collect information across shared library boundaries. If all you want is data from one library, use one of the static link libraries and link it into your project.

For example, say you split your application into an application and 2 shared libraries and want to profile it. You can link a static link library to each part and get three reports, one for each piece. But that data is very hard to interpret because each report is missing some context that happens to be included in another report. By using the shared library version of the profiler, you will get one report that spans all three components.

---

**TIP** We have some more tips! When you use a PPC multiprocessor library for profiling, add

```
#include <ProfilerMPHelper.h>
```

to any file that calls `MPWaitOnQueue()`, `MPWaitOnSemaphore()` or `MPEnterCriticalRegion()`. This helps the profiler adjust the timings from the main task. MP Profiling only works for version 1.4 of the MP software.

---

The profiler also has separate interface files for C/C++ and Pascal. They are listed in Table 5.3.

**Table 5.3 Profiler interface files**

<code>profiler.h</code>	The header file for the profiler API for C and C++. Include this file to make calls to control the profiler.
<code>profiler.p</code>	The interface file for the profiler API for Pascal. Include this file to make calls to control the profiler.

---

## Profiling Special Cases

The profiler handles recursive and mutually recursive calls transparently. The profiler also warns you when profiling information was lost because of insufficient memory. (The profiler uses memory buffers to store profiling data.)

For leading-edge programmers, the profiler transparently handles and reliably reports the times for asynchronous completion routines, multiple threads, 68K code resources, PowerPC shared libraries, and even abnormally terminated routines exited through the C++ exception handling model (try, throw, catch) or the ANSI C library `setjmp()` and `longjmp()` routines.

C++ programmers may wish to consult the Metrowerks PowerPlant documentation for information on the `StProfileSection` class. This class automates the process of initializing the profiler, dumping results, and exiting the profiler. You do not need to use the rest of PowerPlant to take advantage of the `StProfileSection` class.

This section describes the special cases that are specific to profiling Mac OS code:

- Profiling Code with `#pragma` Statements
- Initializing Profiler with `ProfilerInit()`
- Terminating Profiler with `ProfilerDump()`
- Profiling Threads
- Viewing Threads in MW Profiler
- Profiling Shared Libraries
- Profiling Code Resources
- Profiling C++ Static Constructors
- Profiling MacApp
- Profiling Asynchronous Routines
- Profiling Abnormally Terminated Functions
- Using the PowerPC PEF Linker
- Debugging Profiled Code

## Profiling Code with #pragma Statements

You can substitute #pragma statements for profiler API function calls to profile your C/C++ code on the function level. However, this is not as useful as the profiler calls. See “Routine-level profiling” for more information.

Setting the “Generate Profiler Calls” Processor preference option sets a preprocessor variable named `__profile__` to 1. If profiling is off, the value is zero. You can use this value at compile time to test whether profiling is on.

Instead of, or in addition to, setting the option in the Processor preferences, you can turn on profiling at compile time. The C/C++ compiler supports three preprocessor directives that you can use to turn compiling on and off at will.

<code>#pragma profile on</code>	enables calls to the profiler in functions that are declared following the pragma
<code>#pragma profile off</code>	disables calls to the profiler in functions that are declared following the pragma
<code>#pragma profile reset</code>	sets the profile setting to the value selected in the preferences panel

You can use these directives to turn profiling on for any functions you want to profile, regardless of the settings in the Processor preferences. You can also turn off profiling for any function you don’t want to profile.

## Initializing Profiler with `ProfilerInit()`

At the beginning of your code, you call `ProfilerInit()` to initialize the Profiler. Table 5.4 shows the prototypes for `ProfilerInit()` for C/C++.

**Table 5.4 ProfilerInit() prototypes**

C/C++	<pre>pascal OSErr ProfilerInit(     ProfilerCollectionMethod method,     ProfilerTimeBase timeBase,     short numFunctions, short     stackDepth);</pre>
-------	--

The parameters tell the profiler how this collection run is going to operate, and how much memory the profiler should allocate for its data buffers. Each parameter and its purpose is given in Table 5.5.

**Table 5.5 ProfilerInit() parameters**

Parameter	Purpose
method	collect detailed or summary data
timeBase	time scale to use in measurements
numFunctions	maximum number of routines to profile
stackDepth	approximate maximum depth of deepest calling tree

The collection method may be either `collectDetailed` or `collectSummary`. If you collect detailed data, you get information for the calling tree—the time in each routine and each of its children in the calling hierarchy. Summary data collects data for the time spent in each routine without regard to the calling chain. Collecting detailed data requires more memory.

The `timeBase` may be—in order of decreasing precision—`bestTimeBase`, `PPCTimeBase`, `microsecondsTimeBase`, `timeMgrTimeBase`, or `ticksTimeBase`. The `bestTimeBase` option automatically selects the most precise timing mechanism available on the computer running the profiled software. `PPCTimeBase` is only available with PowerPC chips.

The `numFunctions` parameter is the approximate number of routines to be profiled. The `stackDepth` parameter is the approximate maximum depth of your calling chain. You don't need to know the precise values ahead of time. If the profiler runs out of

memory to hold data in its buffers, it loses some data but you'll be told in the results. You can then modify the parameters in the call to `ProfilerInit()` to increase the buffers and preserve all your data.

The profiler allocates buffers in temporary memory based on the method of collection, the number of routines, and the depth of the calling tree. Using temporary memory reduces the effect that the profiler has on the application's memory partition.

The call to `ProfilerInit()` returns a non-zero error value if the call fails for any reason. Use the return value to ensure that memory was allocated successfully before continuing with the profiler. Typically you would add this call as conditionally compiled code so that it compiles and runs only if profiling is on and the call to `ProfilerInit()` was successful.

You call `ProfilerInit()` before any profiling occurs. Typically you make the call at the beginning of your code.

**See also** "Time and Timebases" and "Memory Usage."

### Calling ProfilerInit() in C/C++

In C/C++, the call would be at the beginning of your `main()` function.

The call might look like this:

```
if (!ProfilerInit(collectDetailed, bestTimeBase, 20, 5))
{
// your profiled code
}
```

Of course, your parameters may vary depending upon how many routines you have and the depth of your calling chains.

**See also** "Profiling C++ Static Constructors" on page 50.

### Calling ProfilerInit() in Pascal

Call `ProfilerInit()` at the beginning of your application. In Pascal, it might look like this:

```
IF ProfilerInit(collectDetailed,
  bestTimeBase, 20, 5) = noErr THEN
BEGIN
```

```
{ your profiled code }  
END
```

Of course, your parameters may vary depending upon how many routines you have and the depth of your calling chains.

## Terminating Profiler with ProfilerDump()

The profiler dumps its data to a file when you call `ProfilerDump()`. The file appears in the current default directory, usually the project directory.

You provide a file name when you call `ProfilerDump()`. You may dump results as often as you like. You can provide a different file name for intermediate results (if you have multiple calls to `ProfilerDump()`), or use the same name. If the specified file already exists, a new file is created with an incrementing number appended to the file name for each new file. This allows the dump to be called inside a loop with a constant file name. This can be useful for dumping intermediate results on a long task.

`ProfilerDump()` does not clear accumulating results. If you want to clear results you can call `ProfilerClear()`.

A typical call to `ProfilerDump()` would be placed just before you exit your program, or at the end of the code you are profiling. The prototypes for `ProfilerDump()` are listed in Table 5.6.

**Table 5.6 ProfilerDump() prototypes**

C/C++	<code>OSErr ProfilerDump(     unsigned char *filename);</code>
-------	--

### Calling ProfilerDump()

There is only one parameter. It must be a pointer to a Pascal-style string that becomes the file name for your results. A typical call might look like this:

```
error = ProfilerDump("\pMyCode.prof");
```

**WARNING!** Calls to `UnloadSeg()` in a 68K application cause serious problems for the profiler. The profiler maintains pointers to locations in the

code. If you unload a segment that has profiled routines, that code is free to move around in memory, or be purged. At that time the profiler's internal pointers become invalid. This is the leading cause of corrupted profile data files.

---

## Profiling Threads

You may use the profiler to study cooperative threads. The profiler does not support preemptive threads.

To profile a cooperative thread, you do a little setup work when you create the thread. You call `ProfilerCreateThread()` to tell the profiler to allocate the necessary buffers. (This doesn't actually create a thread, just the necessary structures so the profiler can keep track of a thread.) You provide three parameters. You specify the number of routines, the byte size of the stack created for the thread, and provide the address where the profiler can return a reference to this thread's profile data. You must keep track of this reference for use in other thread-related profiler calls.

When your thread gains control and executes, its `SwapIn` callback routine is called. In that routine, call `ProfilerSwitchToThread()` to tell the profiler that this thread is now running. You provide the profiler's thread reference you got when you called `ProfilerCreateThread()`.

---

**NOTE** Do not profile the `swapIn` routine itself. Thread callback procs (`swapIn` & `swapOut` for example) should be compiled to not call the profiler.

---

When your thread finishes, call `ProfilerDeleteThread()`. You provide the profiler's thread reference to identify which thread is involved. This call cleans up the profiler's thread-tracking structures. You do not need to dump your data before calling `ProfilerDeleteThread()`. The data collection buffers are separate from the memory used to track each thread.

You do not need to call `ProfilerCreateThread()` for the implicit main thread created for your process. However, if you profile code in some other thread and then want to profile the main

thread, you will need to call `ProfilerSwitchToThread()` to resume profiling the main thread. You can get the necessary thread reference by calling `ProfilerGetMainThreadRef()`.

**TIP** If you are programming in C++, you may want to consider using the PowerPlant `LThread` class. This class has built-in support for profiling threads. See the PowerPlant source code and documentation for more details on this class.

---

## Viewing Threads in MW Profiler

If you are profiling a multi-threaded application, you should be aware of how the results are shown in MW Profiler.

In summary mode, the profiler will mix all the threads together, and you can't tell what is called from what thread. Object mode does the same as summary mode.

In detailed mode, if the threads have a different top-level procedure (the procedure you register with the Thread Manger), then the profiler will show them as separate top-level entries. Threads with the same top-level name will be combined together. For example, say you have a program whose `main()` initializes the profiler, starts up 3 threads, and then calls a function to run event loop.

### Listing 5.1 Profiling multi-threaded code

```
main()
{
    ProfilerInit(collectDetailed,bestTimeBase,20,5);

    //...Create a DoBackGroundWork thread
    //...Create another DoBackGroudWork thread
    //...Create a thread that runs DoIdleStuff

    DoMainEventLoop();

    ProfilerDump(`test");
    ProfilerTerm();
}
```



The profile output file for this will have three top-level functions listed in detail mode: `DoBackgroundWork`, `DoIdleStuff`, and `DoMainEventLoop`. The two threads that are running `DoBackgroundWork` would combine their profile information.

## Profiling Shared Libraries

The profiler supports profiling shared libraries for PowerPC and CFM 68K.

Add the shared library named `ProfilerLIB` to your project instead of the static link library `ProfilerPPC.lib`. Include the same header (`profiler.h`).

The shared library profiler collects data from an application and any shared libraries simultaneously. This allows you to split an application into multiple shared libraries and a smaller application, but still profile as if all the components were one large application. Each shared library you create must be compiled with **Generate Profiler Information** selected in the Code Generation pane in the Project Settings dialog.

---

**NOTE** If you are compiling code for the PowerPC, click the **Profiler Information** checkbox in the Code Generation pane.

---

See also “Turn on profiling.”

## Profiling Code Resources

The profiler supports profiling code resources.

For 68K code resources, add the `Profiler68kA4.lib` library to your project instead of `Profiler68k.lib`. Include the same header (`profiler.h`).

The code resource must not move in memory between the `ProfilerInit()` and `ProfilerTerm()` calls. This is because the A4 globals can move. The profiler has a pointer to them that is not updated except when you call `ProfilerInit()`.

For PowerPC code resources, you may use either the static link library `ProfilerPPC.lib` or the shared library `ProfilerLIB`. You must

make sure that the PowerPC code resource does not move in memory while profiling. This means that profiling accelerated resources under applications that unlock the code resource after each call is not supported.

## Profiling C++ Static Constructors

It is possible to profile static constructor calls if the profiler is initialized before the static constructor is called. If you are working with shared libraries, you need to determine the order in which the shared libraries are loaded. Put the initialize call to the profiler in the first one loaded.

To profile the static constructors called from `__sinit()`, the profiler must be initialized before calling `__sinit()`.

To do this with the 68K compiler, use the function `__PreInit__ (void)` declared in Listing 5.2.

### Listing 5.2 `__myinit`: Initializing the 68K compiler in C++

```
extern "C" void __PreInit__(void);
void __PreInit__(void)
{
    OSErr error;
    error=ProfilerInit(collectDetailed,
                     bestTimeBase, 20, 5);
}
```

Of course, you'll set the parameters in the call to `ProfilerInit()` appropriately for your work.

For PowerPC code, write a small routine `__myinit` that initializes the profiler, then calls `__sinit`. Put `__myinit` in the initialization routine in the Linker preferences pane in the Project Settings dialog.

### Listing 5.3 `__myinit`: Initializing the PPC compiler in C++

```
OSErr __myinit(InitBlockPtr initBlock)
{
    OSErr error;

    error=ProfilerInit(collectDetailed,
                     bestTimeBase, 20, 5);
}
```

```
if (error == noErr)
    __sinit();
return error;
}
```

## Profiling MacApp

Before profiling a MacApp-based 68K project, go through the MacApp source code and comment out all calls to `UnloadSeg()`. There are only a few.

In a 68K project, calls to `UnloadSeg()` cause serious problems for the profiler. The profiler maintains pointers to locations in the code. If you call `UnloadSeg()` on a segment that has profiled routines, that block of code is free to move around in memory, or be purged. At that time the profiler's internal pointers become invalid. This is the leading cause of "corrupted" profile data files.

## Profiling Asynchronous Routines

The CodeWarrior profiler supports profiling asynchronous completion routines, VBL tasks, Time Manager tasks, deferred tasks, and related code. This is essentially transparent and a by-product of the internal structure and design of the profiler.

However, there are some things to be aware of when working with asynchronous routines and other code that runs at interrupt time.

- Don't try to call the profiler in a socket listener because the profiler adds too much latency for both LocalTalk and Ethernet devices.
- Be aware that running the profiler at interrupt time may cause problems because of the increased time that interrupts are disabled.
- Use the large code model for 68K code resources.

The small and smart code models for 68K can cause problems for completion routines. They use A5 relative addressing to call the profiler. Because A5 is not always set up when a completion routine is called, the embedded call to the profiler will not work. The large code model uses absolute addressing to call the profiler, which avoids the problem.

Another way around this problem is to have a small stub routine that sets up A5 and calls your routine that does the real work. The stub routine should be compiled without a call to the profiler, and the real routine can call the profiler.

## Profiling Abnormally Terminated Functions

The profiler correctly reports data for abnormally terminated functions that exited through the C++ exception handling model (try, throw, catch) or the ANSI C library `setjmp()` and `longjmp()` routines. You do not have to do anything to get this feature, it is automatic and part of the profiler's design.

However, there is a possibility of some errors in the reported results for an abnormally terminated function.

First, the profiler does not detect the abnormal termination until the next profiling call after the abnormal termination. Therefore, some additional time will be reported as belonging to the terminated function.

Second, if the next profiler event is a profiler entry, and the new stack frame for that function is larger than the frames that were abnormally exited, the profiler will not immediately detect that the original function was abnormally terminated. In that case the profiler will treat the function just entered as a child of the function abnormally terminated. The profiler will correct itself on the next profiling event without this property—that is, when the stack returns to a point smaller than it was when the abnormally terminated function exited.

Finally, remember that the profiler is not closed properly and the output file is not dumped when `ExitToShell()` is called. If you need to call `ExitToShell()` in the middle of your program and want the profiler output, call `ProfilerDump()`.

If you are using the profiler, you must always call `ProfilerTerm()` before `ExitToShell()`.

**WARNING!**

If a program exits after calling `ProfilerInit()` without calling `ProfilerTerm()`, timers may be left running that could crash the machine.

## Using the PowerPC PEF Linker

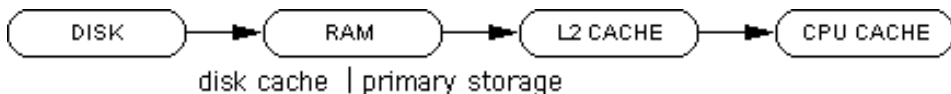
If you have a PowerPC target, there is one final optimization you can perform to make your code run faster. Sorry, but this optimization does not apply to 68K programming!

The way in which code is organized in memory has a subtle influence on its execution speed. Some functions frequently refer to one another. If we can organize memory so that these functions are placed close together, we can improve the hit rates for the instruction cache. Maximizing the hit rate of the cache makes memory access, and thus our program, faster.

Calling code up from disk is slower than calling it up from RAM. By grouping functions together, the code may live in RAM for a longer time and reduce the amount of time spent on disk access.

Though it is possible that some of your functions may be too large to exist in the cache at the same time, this is less likely these days on systems with an L2 cache of 1MB or more. Figure 5.1 illustrates how your code makes its way from the disk to your CPU.

**Figure 5.1** Movement of code from disk to CPU



### Generating a Link Arrangement File

To optimize the way our code is loaded and executed at runtime, we can ask the profiler to generate a link arrangement (`.arr`) file. The `.arr` file groups the functions in our application by their call frequency—functions that refer to each other often will be clustered together. CodeWarrior uses the data in the `.arr` file to generate a faster executable.

The two options available to you are **Generate Arrange File** which uses a less-sophisticated depth-first traversal algorithm, and **Generate Weighted Arrange File** which incorporates true call frequency ordering. For general use, you will find that the call frequency ordering of a **Weighted Arrange File** provides the best function arranging.

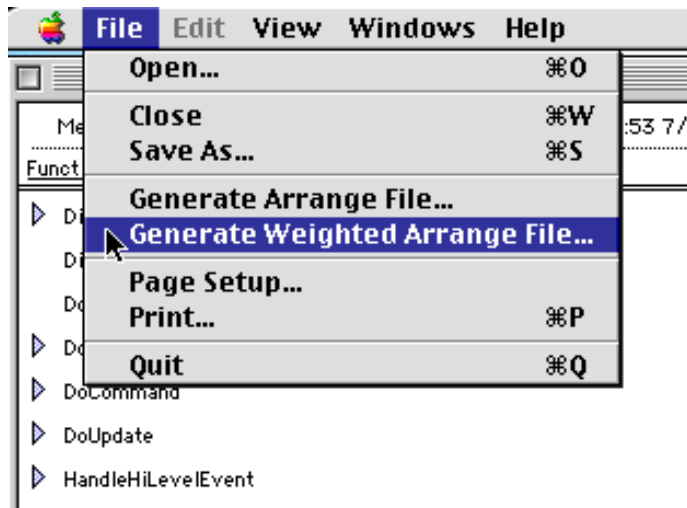
To generate and use a **Weighted Arrange File**, follow these steps:

1. **Load the profiler data file into the Profiler.**
2. **Generate the .arr file.**

Select **File > Generate Weighted Arrange File** from the Profiler menu as shown in Figure 5.2. The name you choose for your link arrangement file must end with the .arr extension. MyApp.arr will do nicely.

The Profiler may spend several minutes generating .arr file for a large application. However, a small application will only take a few seconds.

**Figure 5.2** Generating the .arr file

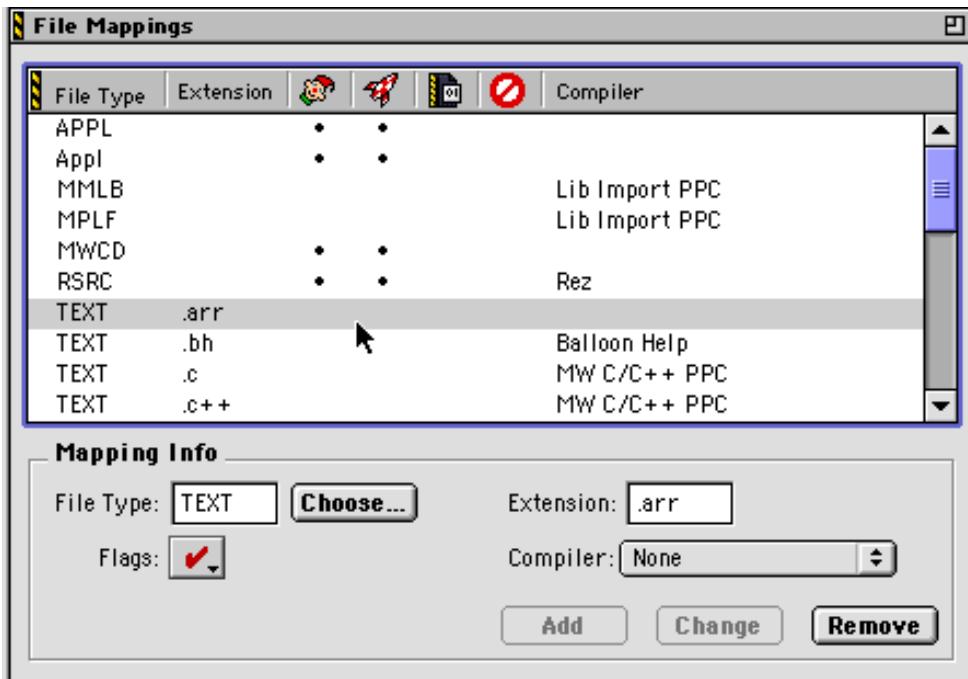


3. **Return to your project in the CodeWarrior IDE.**

4. Add the .arr extension to File Mapping.

CodeWarrior should already be set up with the .arr file mapping for new projects by default. If you converted an older project without the file mapping, make your project aware of your .arr file by adding its extension to the **File Mappings** panel. In **Target Settings > File Mappings**, create a new mapping for the .arr type. As shown in Figure 5.3, you should map the .arr extension as a TEXT file, with no active flags or compiler settings.

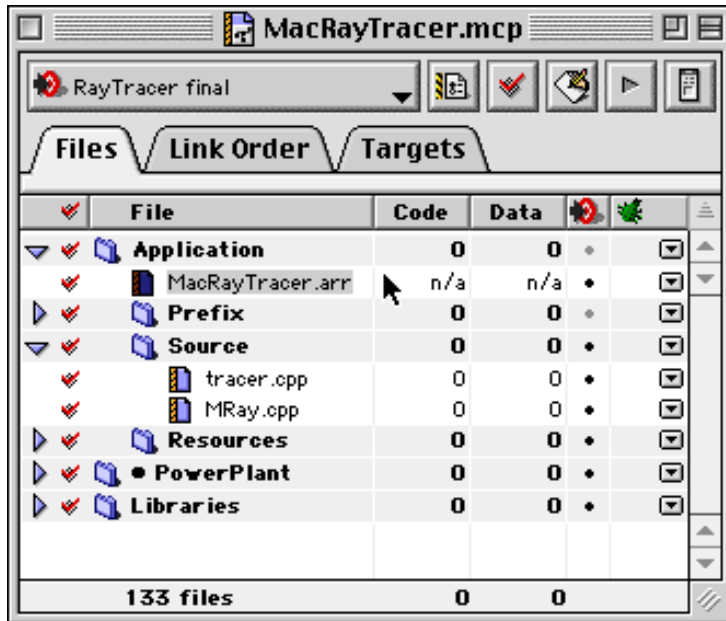
Figure 5.3 File Mappings for the .arr extension.



5. Add the .arr file to your project.

Drag MyApp.arr into your project window as shown in Figure 5.4.

Figure 5.4 Adding the .arr file to your project.

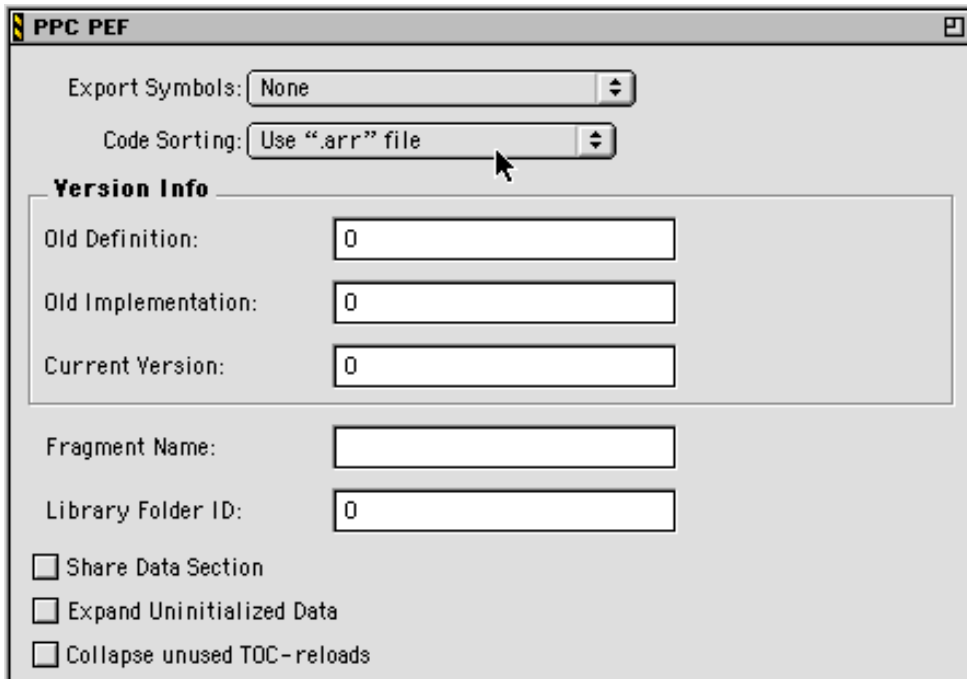


6. Activate the .arr file in the PPC PEF panel.

Go to the **Target Settings > PPC PEF** panel and click the **Code Sorting** check box. Choose **Use “.arr” file** from the menu choices available, then save your settings.



Figure 5.5 Configuring the PPC PEF to use the .arr file.



#### 7. Build your application.

The application that CodeWarrior builds will be optimized to provide you with the fastest memory accesses possible. If you add new functions or function calls to your application in the future, you need to generate a new .arr file.

Do not be alarmed if an error message such as the following appears:

```
Link Warning : sort file 'MyAppName.arr' did not
list all code symbols
```

This warning is normal and should be expected. When the profiler generates the .arr file, it only includes functions and symbols that are profiled. For instance, MSL library calls will not appear in the .arr file because the MSL is compiled with profile information off. The warning message appears to notify you of this.

You may disable these warnings by checking the **Suppress Warning Messages** checkbox in the PPC Linker panel.

**TIP** When using the profiler to optimize your code, we recommend that you create separate targets; include your profiler libraries in one, but not the other. You can then use the profiler target to generate the .arr file, and then drop the .arr file into your streamlined release target.

---

## Debugging Profiled Code

It is possible to debug code that has calls to the profiler in it. However, the profiler does interfere with stepping through code. You may find it simpler to debug non-profiled code, and profile separately. In this section, We'll take you through what happens when you step into a profiled routine and step out of a profiled routine. In addition, we'll talk about the effect that stopping in the debugger has on the profile results.

**See also** the *CodeWarrior IDE User Guide* for more information on how to use the debugger.

### Stepping into a profiled routine

If you step into a profiled routine you'll see assembly code instead of source code. The compiler has added calls to `__PROFILE_ENTRY` at the start of the routine. This is how the profiler knows when to start counting time for the routine.

If you step through the assembly code far enough to get to the code derived from the original source code, then switch the view from source to assembly and back again, you can see the original source code.

### Stepping out of a profiled routine

If you single-step out of a routine being profiled, you end up in the `__PROFILE_EXIT` assembly code from the profiler library. This is how the profiler knows when to stop counting time for the routine.

### Effect of stopping on the profile results

If you stop in a profiled routine, the profiler counts all the time you spend in the debugger as time that routine was running. This skews the results.

**WARNING!** If you debug profiled code, you must be careful not to kill the code from the debugger. Remember, if you have called `ProfilerInit()` you must call `ProfilerTerm()` on exit or you may crash the computer.

---

## Profiler Limitations

For application programming, the CodeWarrior profiler has only one limitation. You must not call `UnloadSeg()` in a 68K application.

In a 68K project, calls to `UnloadSeg()` cause serious problems for the CodeWarrior profiler. The profiler maintains pointers to locations in the code. If you call `UnloadSeg()` on a segment that has profiled routines, that block of code is free to move around in memory, or be purged. At that time the profiler's internal pointers become invalid. This is the leading cause of "corrupted" profile data files.

**See also** "Profiling Special Cases" on page 42 for additional considerations in non-application projects.



# Profiling in PowerPlant

---

The PowerPlant Profiler Class is a wrapper class for the Profiler API. It provides an easy to use interface for the initialization and termination functions of the profiler.

## Introduction to Profiling in PowerPlant

If you have used the profiler before to profile C or Pascal code, you are already familiar with how the initialization and termination routines work. In PowerPlant, you never need to call these routines directly. Instead, these calls are handled for you by the constructor and destructor of the profiling class.

In this chapter, you'll learn:

- Profiling Strategy—a brief review of profiling in general
- Profiling Class—shows you how the profiling class is set up and what is required to use it
- Profiling Your Code—a short description on how to profile your code

There is a code exercise provided at the end of this chapter to give you a full working tutorial on how to use the profiler class in your PowerPlant programs.

## Profiling Strategy

Before you delve into the depths of the Profiler Class, you should familiarize yourself with the profiler in general. We recommend that you read the *Profiler Guide* first to become familiar with all aspects of the profiler. Some information is reiterated here, but you should read the *Profiler Guide* for more in-depth information.

You use a profiler to measure the runtime performance of your code. What is usually important is how your code's performance measures up to some standard. When approaching the problem of measuring performance, you might want to take these three steps.

**1. Establish your standards.**

For example, you might decide that you want the program to load in less than ten seconds, or check the spelling of a five-page document that contains no misspellings in 15 seconds. Decide the platform you will use for testing as well: a Power Macintosh is a bit faster than a Mac Plus.

**2. Determine how to measure time.**

Your measurement device may be no more complicated than a stopwatch, or you may need to add some simple code to count ticks. At this phase you want to test the code in as close to its finished form as possible, so measure time in a way that is accurate enough to suit your needs, and that has the lowest impact on your code's natural performance. You do not want to run a full-blown profile here, because profiling can add significant overhead, thus slowing down your code's raw performance.

**3. Run the tests and measure results.**

If you meet your performance goals, your job is done. If your code does not meet your goals, then it's time to profile your code.

## Rules of Thumb

To make sure you get a good profile of your code, there are few general rules of thumb you need to follow.

**1. Turn off all optimizations**

Optimization settings will cause skewed results in the profiler output. Part of the reason for profiling your code is so you can optimize it yourself as much as possible. Once you are confident you've done all you can, you can turn on optimizations in the final build of your application.

**2. Turn off virtual memory or RAM Doubler**

Profiling code with virtual memory or RAM Doubler active produces erratic and skewed result times. If your program runs quickly without virtual memory or RAM Doubler active, it will run quickly with them active.

### 3. Disable Speed Doubler

Speed Doubler is a performance enhancement series of extensions. As such, it will skew your result times and should therefore be disabled while profiling your code.

---

**TIP** It's actually best to turn off all extensions when doing a profile of your code. This will eliminate any chances of extension problems and background tasks skewing the profile results. Restart your computer with the Shift key held down to disable extensions.

---

### 4. Do not use `UnloadSeg()` in 68K code

Calls to `UnloadSeg()` will cause garbage output when you look at your data with the MW Profiler application. Comment out, or remove calls to `UnloadSeg()` until the final build of your program.

## Profiling Class

The PowerPlant profiler class is called `StProfileSection` and is found in `UProfiler.cp`. The "St" means that this is a stack-based class and functions accordingly. This class is an independent class and can therefore be used with or without the rest of PowerPlant.

`StProfileSection` makes two assumptions when calling `ProfilerInit()`. First, it uses `collectDetailed` as the data collection method. Second, it uses `bestTimeBase` for the timing method parameter. `StProfileSection` only has two member functions, a constructor and a destructor.

**See Also** "Profiler Function Reference" for more information on the parameters used for `ProfilerInit()`.

### StProfileSection

The class constructor takes three parameters and performs two main operations. It initializes three data members with the values you provide, and calls `ProfilerInit()` using these values. The constructor is declared as follows:

```
StProfileSection( Str255 inDumpFileName,  
                Sint16 inNumFunctions,
```

```
SInt16 inStackDepth);
```

**TIP** If you are profiling your entire PowerPlant program, you should use large values for `inNumFunctions` and `inStackDepth`. The values you use will depend on the size of your program. For example, use 2500 for `inNumFunctions` and 100 for `inStackDepth` to start, then increase either of these values if necessary after each profile run. Also, remember it may become necessary to increase the memory size of your application program as profiling takes more memory than would normally be required.

The three data members are shown in Table 6.1.

**Table 6.1** StProfileSection data members

Data Member	Stores
<code>mProfilerDumpFile</code>	pascal string for the dump filename
<code>mNumFunctions</code>	number of routines to create buffer space for
<code>mStackDepth</code>	number of routines deep the stack can get

## ~StProfileSection

`~StProfileSection()` is the class destructor. It is called automatically as soon as the code you are profiling goes out of scope. If you are profiling one area of your code, it's when that function ends. If you are profiling your entire application, it's when the application quits.

The destructor does all the house cleaning work. It calls `ProfilerSetStatus()` to stop profiling while it performs the clean up, calls `ProfilerGetDataSizes()` to test memory requirements and sends an `Assert_()` if there wasn't enough memory, calls `ProfilerDump()` to dump the results to the file, and finally, calls `ProfilerTerm()` to end the profile session.



---

**NOTE** If you get an `Assert_()` message, it means the values you provided for `inNumFunctions` and `inStackDepth` are too low. You will need to re-profile your code using larger values.

---

## Profiling Your Code

To profile your code, you do four things.

- 1. Set up your project for profiling**

This step is very similar to the description given in the *Profiler Guide*. The only changes are the source file you add to your project, and the header file you add to your code. More detail on this is given in the code exercise.

- 2. Run a profiler on the area of the code you want tested.**

This might be a single routine, a group of routines that perform a task, or even the entire application. What you profile depends upon what you are testing.

Declare a local `StProfileSection` variable, and provide the proper setup information. Your declaration will look something like this:

```
StProfileSection theProfile("\pTest", 50, 50);
```

All code that runs from that point on, until `theProfile` goes out of scope, will be profiled.

- 3. Analyze the data collected by the profiler and improve your code.**

You study the results of your profiling and look for problems and room for improvement. For details on how to review the results, see "Viewing Results."

The profiling process is iterative. You repeat steps 2 and 3 until you achieve the performance gain you need to meet your goals.

- 4. Run your code without the profiler**

When you are satisfied that you have reached your goals, you have one more step to perform. You should run your original tests—without the profiler of course—to verify that your code in its natural state meets your performance goals.

Implementing the above steps in your program is simple and only requires two lines of code.

## Code Exercise for Profiling

This code exercise uses the Documents *solution* code from the File I/O chapter of *The PowerPlant Book*. The code can be found on the CodeWarrior Reference CD, in the CodeWarrior Examples: MacOS Examples:PP Book Code:Chap 13 Solution Code folder.

**WARNING!** It is important for this exercise that you use the **Solution** code for the Documents application. Using the Starter Code will either not compile, or crash when run, unless you have finished it in the exercise in *The PowerPlant Book*.

---

In this exercise you will perform two different profiles. The first profile will only focus on one routine (and any routines it calls). The second profile will be the entire application. The steps for each are similar.

### Profiling a Single Routine

There is one important thing to remember when you want to profile a single routine. You are also profiling all other routines called directly or indirectly by the routine being profiled. This is important in order to provide the correct parameters to the constructor. This part of the code exercise will profile the `OpenFile()` routine in `CTextDocument.cp`.

**WARNING!** When you are profiling a single routine, you *must* declare your `StProfileSection` variable in the function that *calls* the routine you want to profile. If you declare an `StProfileSection` variable *inside* the function you want to profile, you will not see the name of your routine in the MW Profiler application. Your routine would still be profiled, but you would be missing an important point of reference in the resulting data.

---

#### 1. Set up the project for profiling

There are two things you need to do for this step. You can do these in any order you wish. First, add the necessary source and library

files to your project. Then turn on profiling from the preference panel. These steps are the same for both 68K and PowerPC projects.

**a. Adding files to the project**

Add `UPProfiler.cp` to your project as well as the appropriate library. The library you add depends on the type of program you are writing and your target platform (68K or PowerPC). Table 6.2 lists the libraries and shows you when to use them.

**Table 6.2 Profiler libraries**

Library	Processor	When To Use
Profiler 68k.Lib	68K (A5 globals)	near/small model applications
Profiler Fa(68k).Lib	68K (A5 globals)	far/large/smart model applications
Profiler 68k.A4.Lib	68K (A4 globals)	near/small model code resources
Profiler Fa(68k.A4).Lib	68K (A4 globals)	far/large/smart model code resources
Profiler CFM68k.Lib	68K (A5 globals)	code fragments and shared libraries
Profiler PPC.Lib	PowerPC	PowerPC applications and shared libraries
Profiler PPC.MP.Lib	PowerPC	PowerPC multiprocessor applications and shared libraries
ProfilerMPLib	PowerPC	PowerPC multiprocessor code fragments and shared libraries.
ProfilerLib	CFM 68K & PowerPC	code fragments and shared libraries

# Freescale Semiconductor, Inc.

Profiling in PowerPlant  
Profiling a Single Routine

---

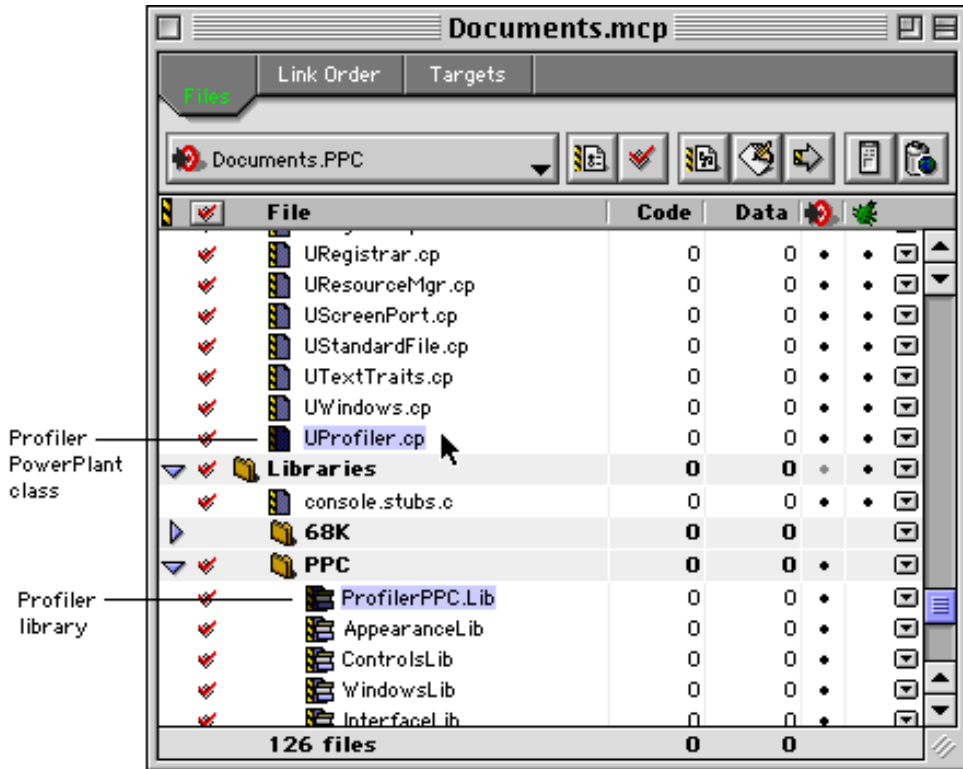
Library	Processor	When To Use
Profiler Carbon.Lib	PowerPC (Carbon)	PowerPC applications and shared libraries for use on Mac OS X or Mac OS 8.1 and above with the Carbon libraries
ProfilerCarbonLib	PowerPC (Carbon)	PowerPC code fragments and shared libraries for use on Mac OS X or Mac OS 8.1 and above with the Carbon libraries

---

The example shown in Figure 6.1 uses `ProfilerPPC.lib` for this exercise.

**See Also** “Profiler Libraries and Interface Files” for more information on profiler libraries.

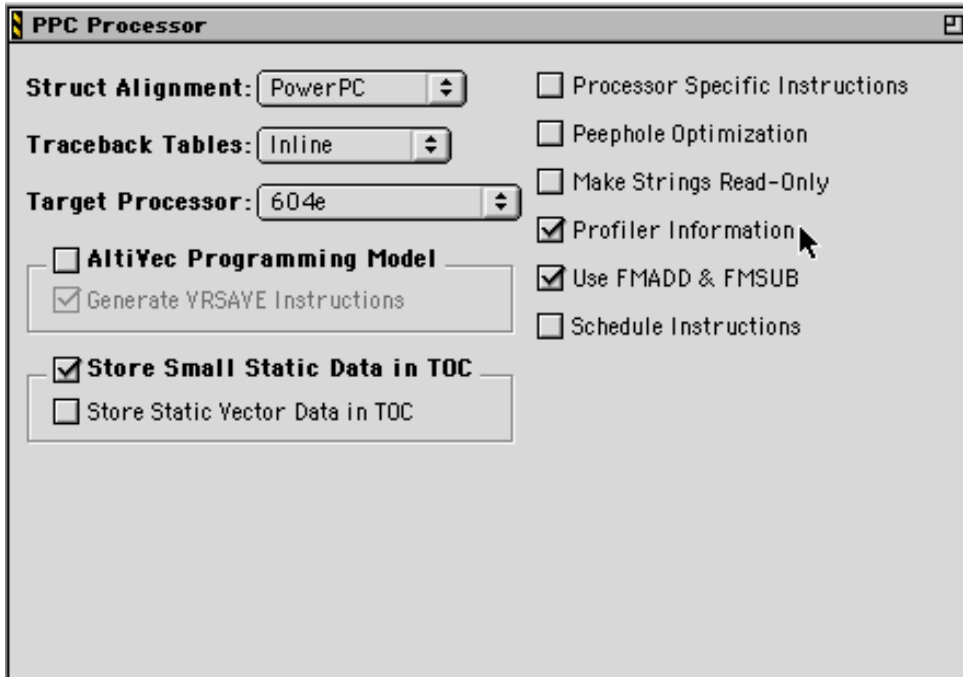
**Figure 6.1 Adding profiler files to the project**



**b. Turn on profiling for the project**

Select **Profiler Information** from the **PPC Processor** preference panel as shown in Figure 6.2.

Figure 6.2 Generate profiler calls



## 2. Add the header file

`CTextDocument.cp`

You need to include `UProfiler.h` in the file that makes the call to the `StProfileSection` class. It is not needed in any other file in your project even if the routine your profiling calls routines in a different file.

```
#include <UProfiler.h>
```

## 3. Declare an `StProfileSection` variable

`CTextDocument()` `CTextDocument.cp`

In order to profile the routine you want, you declare the `StProfileSection` variable in the routine that makes the call to the function you are concerned with. In this case, you put the declaration in the class constructor right before the call to `OpenFile()`.

```
// Set name of window or open file.
if ( inFileSpec == nil ) {

    NameNewDoc();

} else {
    // Profile the OpenFile routine
    StProfileSection theProfile("\pTest", 50,
50);
    OpenFile( *inFileSpec );
}
}
```

#### 4. Build and run the application

That's all there is to it! Once the application builds successfully, run the application, open a file, then quit. There will be a file called "Test" in the same folder as the application. You can then use the MW Profiler application to view the results.

**See Also** "Viewing Results" for more information on how to view your data.

## Profiling an Application

You might decide you want to get the "big picture" of what's happening in your code. To do that, you would profile your entire PowerPlant program.

The steps you perform are exactly the same as shown above with the exception of steps two and three. If you've performed the above tasks, remove the header file and the declaration of `StProfileSection` and replace them with the following steps.

#### 1. Add the header file

CDocumentsApp.cp

Add the header file to the top of CDocumentsApp.cp.

```
#include<UProfiler.h>
```

#### 2. Declare the `StProfileSection` variable

```
main()                                CDocumentsApp.cp
```

You do not necessarily need to profile the initialization code unless you feel it is a concern. The code below declares the class before the `Run()` command is given. You may put it at the top of `main()` if you wish.

```
// Create the application object and run it.  
CDocumentsApp theApp;  
StProfileSection theProfile("\pApp", 2500,  
100);  
theApp.Run();
```

### 3. Build and run the application

That's all there is to it! Once the application builds successfully and runs, a complete profile of the application will be built. You have to quit the application for the `StProfileSection` class to go out of scope, write the data, and close the file. Once you've done that, there will be a file called "App" in the same folder as the application. You can then use the MW Profiler application to view the results.

---

**TIP** If you run multiple profiles of your code, the profiler routines are smart enough not to overwrite the same dump file. Instead, it will append a number after the file name (e.g.: App2). This is useful if you want to compare different profiles of your program (as you make changes) and not have to change your code. For example, you may want to compare the 68K version with the PowerPC version. For reliable comparisons, make sure you profile *exactly* the same routines each time, and in each version.

---

**See Also** "Viewing Results" for more information on how to view your data.



# Troubleshooting

---

This chapter contains frequently asked questions (and answers) about the Profiler. If you have a problem with the profiler, come here first. Others may have encountered similar difficulties, and there may be a simple solution.

The general topics that are covered include:

- Profile Times Vary Between Runs
- Problems while Profiling Inline Functions
- Profiling Library Could not be Found
- Profiler and Virtual Memory
- Names are Garbled when Viewing a 68K Profile Dump

## Profile Times Vary Between Runs

### Problem

I'm getting different results (within 10%) in MW Profiler every time I run my program.

### Background

There are two potential reasons that this may be happening. Both are time-related problems. The first problem that can occur is inadequate time in the function relative to the profiler resolution. The second problem is clock resonance.

#### ***Inadequate time in the function***

If the function time that you are trying to measure is only 10 times greater than the resolution of the timebase you are using, you'll have this problem.

The profiler uses these timebases:

Name	Resolution	Code
Ticks	~16700 $\mu$ s	68K and PPC
Time Manager	~ 20 $\mu$ s	68K and PPC
Microseconds	~ 20 $\mu$ s	68K and PPC

### Solution

To solve this problem, increase the number of times your function is called, then the average the profiler computes will be more accurate.

Sometimes it is helpful to pull a routine out of a program, and into a special test program which calls it many times in a loop for performance tuning purposes. However, this technique is susceptible to cache differences between the test and real program.

### **Clock resonance**

If the operations you are performing in your profiled code coincide with the incrementing of the profiler clock, the results can be distorted, and could show wild variations.

### Solution

One way to avoid this problem is to increase the number of times your function is called.

Another way to avoid this problem is to increase the accuracy of the clock the profiler is using. You can only do this with the PowerPC profiler. The PowerPC profiler can use the RTC or TB registers of the PowerPC chip which have 128 ns or better resolution. Make a PowerPC version of your project and use `bestTimeBase` in your call to `ProfilerInit()`.

## Problems while Profiling Inline Functions

### Problem

My inline functions are not getting inlined when I'm profiling my code. What's happening?

## Background

When the compiler switch for profiling is turned on, the default setting for “don’t inline functions” is changed to true. This is so that these functions will have profiling information collected for them.

## Solution

Place a `#pragma dont_inline off` in your source file to turn on function inlining again. You will not collect profile information for inline functions. In effect, a function can be inlined or profiled, but not both. The profiler cannot profile an inlined function.

---

**TIP** If you use the `#pragma dont_inline off` in your code, you may see profile results for some inline functions. When you declare an inline function, the compiler is allowed, but not required to inline the function. It is perfectly legal for the compiler to inline some functions, but not others. Data is collected only for the calls that were not inlined. The calls that were inlined have their time added into the time of the calling function.

---

## Profiling Library Could not be Found

### Problem

While trying to profile my dynamically linked library (shared library), I get an error message saying that the profiling library could not be found.

### Background

This problem occurs when trying to use the profiling library to profile your dynamically linked library and the profiling library is not in the search path.

### Solution

Add the profiling library to the search path. If you are using the CodeWarrior IDE, see the *CodeWarrior IDE User’s Guide* for information on search paths. If you are using MPW, see your *MPW User’s Manual*.

ProfileLib is a shared library and should be in your Extensions folder. Make sure a path to the Extensions folder is set for your project.

## Profiler and Virtual Memory

Profiling code with virtual memory or RAM Doubler active produces erratic and skewed result times. Make sure virtual memory or RAM Doubler are disabled when profiling your code.

---

**TIP** It's actually best to turn off all extensions when performing a profile of your code. This will eliminate any chances of extension problems and background tasks skewing the profile results. Restart your computer with the Shift key held down to disable extensions.

---

## Names are Garbled when Viewing a 68K Profile Dump

### Problem

I see garbage names in MW Profiler on 68K (non-CFM) profile runs.

### Background

You've left calls to `UnloadSeg()` in your program. Calls to `UnloadSeg()` in a 68K application cause serious problems for the profiler. The profiler maintains pointers to locations in the code. If you unload a segment that has profiled routines, that code is free to move around in memory, or be purged. At that time the profiler's internal pointers become invalid. This is the leading cause of corrupted profile data files.

### Solution

Remove, or comment out, the calls to `UnloadSeg()` when profiling your code.

# Profiler Reference

---

This chapter contains the detailed technical reference information you may need when using the profiler.

The topics discussed include:

- Compiler Directives—handling compiler directives in Mac OS code
- Memory Usage—understanding memory usage in Mac OS code
- Time and Timebases—the available time resolutions for Mac OS code
- Profiler Menu Reference—a reference for the main menus in MW Profiler
- Profiler Function Reference—a reference for all of the profiler API functions

## Compiler Directives

You can control routine-level profiling using compiler directives.

The C/C++ compiler supports three preprocessor directives that you can use to turn compiling on and off at will.

<code>#pragma profile on</code>	enables calls to the profiler in functions that are declared following the pragma
<code>#pragma profile off</code>	disables calls to the profiler in functions that are declared following the pragma
<code>#pragma profile reset</code>	sets the profile setting to the value selected in the preferences panel

You can use these directives to turn profiling on for any functions you want to profile, regardless of the settings in the Processor preferences. You can also turn off profiling for any function you don't want to profile.

## Testing for the Profiler

As there are compiler directives to turn the profiler on and off, there are also directives to test if the profiler is on. You can use these tests in your code so that you can run your program with or without the profiler and not have to modify your code each time.

In C/C++, use the `#if-#endif` clause. For example:

```
void main()
{
#ifdef __profile__    // is the profiler on?
    if (!ProfilerInit(collectDetailed, bestTimeBase, 20, 5))
    {
#endif
    test(15);
#ifdef __profile__
    ProfilerDump("\pExample.prof");
    ProfilerTerm();
    }
#endif
}
```

In Pascal, use the following test:

```
{ $IFC OPTION (PROF) }
    if (ProfilerInit(collectDetailed, bestTimebase,
                    100, 20) = noErr) then begin
{ $ENDC }
```

See also "Routine-level profiling."

## Memory Usage

The profiler allocates two buffers in temporary memory to hold data as it collects information about your code: one based on the number of routines, and one based on the stack depth. You pass these parameters in your call to `ProfilerInit()`.

In summary collection mode, the profiler allocates 64 bytes \* numFunctions and 40 bytes \* stackDepth.

In detailed collection mode, the profiler allocates 12 \* 64 \* numFunctions bytes and 40 \* stackDepth bytes.

As an example, assume numFunctions is set to 100, and stackDepth to 10. In summary mode the profiler allocates buffers of 6,400 bytes and 400 bytes. In detailed mode it allocates buffers of 76,800 bytes and 400 bytes.

ProfilerGetDataSizes() lets you query the profiler for the current size of the data collected in the function and stack tables. This information can be used to tune the parameters passed to ProfilerInit().

See also “ProfilerInit()” on page 85.

## Time and Timebases

The CodeWarrior profiler supports four timebases. A timebase is the clock interval used to measure time in a routine. The shorter the interval, the more precise the measurements.

When you call ProfilerInit() you specify the desired timebase. The constant bestTimeBase tells the profiler to figure out the most precise timebase available on your platform and to use it.

Table 8.1 describes each timebase:

**Table 8.1 Profiler timebases**

<b>Timebase</b>	<b>Description</b>
bestTimeBase	This is the preferred timebase. Using this timebase, the profiler selects the timebase with the highest resolution for your machine.

# Freescale Semiconductor, Inc.

## Profiler Reference Profiler Menu Reference

---

PPCTimeBase	This timebase is only available on a Power Macintosh from PowerPC code. It cannot be used in 68k code. It uses the built-in timing facilities of the PowerPC processor. On a 601 it uses the RTC registers. On subsequent PowerPC processors it uses the TB registers. This timebase is very low overhead and high accuracy. If this timebase is specified but not available, <code>ProfilerInit()</code> returns <code>paramErr</code> .
microsecondsTimeBase	This timebase uses the <code>_Microseconds</code> trap to measure time. It has the same accuracy as the <code>timeMgrTimeBase</code> , but less overhead (one trap vs. three traps). This timebase may not be available on all Macintosh computers. If this timebase is specified but not available, <code>ProfilerInit()</code> returns <code>paramErr</code> .
timeMgrTimeBase	This timebase uses the microseconds timing ability of the Time Manager. This is more accurate than the <code>ticksTimeBase</code> , (~20 $\mu$ s), but has more overhead. On a Power Macintosh running System 7.1.2, the traps called run emulated, so there are three mixed mode switches for every routine call. This timebase is available on all Macintoshes with System 7.0 or greater.
ticksTimeBase	This counter increments 60 times a second. It is a very low overhead timebase with coarse accuracy. This timebase is available on all Mac OS computers.

**WARNING!** `PPCTimeBase` does not work correctly under Mac OS X. If you use it, you actually get microseconds timing instead.

---

## Profiler Menu Reference

This reference section discusses the functionality of each menu item in MW Profiler, the data viewing application for the CodeWarrior profiler system.

These are the main menus:

- About MW Profiler—learn about MW Profiler



- File Menu—open, close, save, and print profiler data files, and quits the application
- Edit Menu—provided for compatibility with other applications, no item is functional in MW Profiler
- View Menu—allows you to control the order and appearance of data in the MW Profiler window
- Windows Menu—a list of all open profiler data files

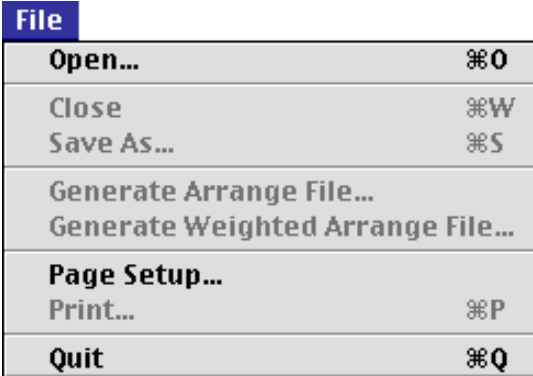
## About MW Profiler

Choosing **About Metrowerks Profiler** from the Apple menu displays copyright and author information about the application.

## File Menu

The **File** menu contains commands to open, close, save, and print documents, and to quit the application, as illustrated in Figure 8.1.

Figure 8.1 The File menu



File	
Open...	⌘O
Close	⌘W
Save As...	⌘S
Generate Arrange File...	
Generate Weighted Arrange File...	
Page Setup...	
Print...	⌘P
Quit	⌘Q

**Open** displays the standard open file dialog that allows you to select and open an existing MW Profiler document.

**Close** closes the active window.

**Save Report As** or **Save As** saves the contents of the active window as a tab-delimited text document. The text file is organized the same way that the window is organized.

**Generate Arrange File** generates a link arrangement file using a depth-first traversal algorithm. The final `.arr` file can then be linked back into your project to produce even faster code.

**Generate Weighted Arrange File** is the same as **Generate Arrange File** but uses a true call frequency ordering algorithm which produces better results. The final `.arr` file can then be linked back into your project to produce even faster code.

**Page Setup** or **Print Setup** displays the standard page setup dialog for the current printer.

---

**TIP** To fit the profiler output across one page, select landscape orientation and 85% reduction.

---

**Print Preview** shows what the data file will look like when you print it out on paper.

**Print** prints the MW Profiler document in accord with the current view and sort order in the window.

**Quit** quits the MW Profiler application.

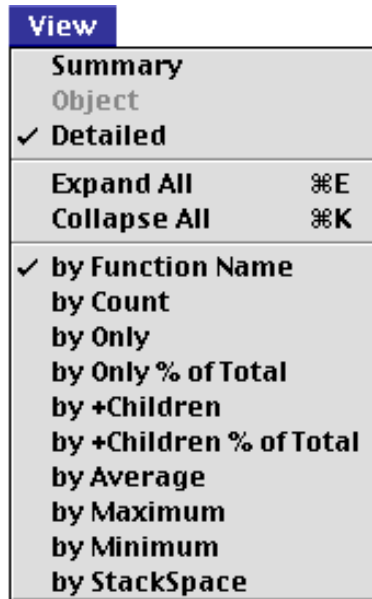
## Edit Menu

All **Edit** menu items are disabled. There is nothing to cut, copy, paste, clear, or select in an MW Profiler window.

## View Menu

The **View** menu contains commands to sort and view MW Profiler data. The currently active option has a check mark in front of the menu item. You may also click on the column header to sort the data by the values in that column. Figure 8.2 shows the options that you can choose.

Figure 8.2 The View menu



- **Summary** displays the active profiler window in summary view. Summary view consists of one row for each routine in the profile data. Summary view is available for any profile.
- **Object** displays the active profiler window in object view. Object view is a summary of each class. Each class can be expanded to show the detail of each method within the class. Object view requires that the profile contain one or more mangled C++ class method names.
- **Detailed** displays the active profiler window in detailed view. Detailed view consists of one row for each path in the call tree. Detailed view requires that the `collectDetailed` method be used.
- **Expand All** expands all rows in an Object or Detailed view.
- **Collapse All** collapses all rows in an Object or Detailed view.
- **by Function Name** sorts the data in the active window by routine name.
- **by Count** sorts the data in the active window by count.
- **by Only** sorts the data in the active window by the Only time.

- **by Only % of Total** sorts the data in the active window sorted by Only percentage. This has the same effect as sorting by Only.
- **by +Children** sorts the data in the active window by +Children time.
- **by +Children % of Total** sorts the data in the active window by +Children percentage. This has the same effect as sorting by +Children.
- **by Average** sorts the data in the active window by Average time.
- **by Maximum** sorts the data in the active window by Maximum time.
- **by Minimum** sorts the data in the active window by Minimum time.
- **by Stack Space** sorts the data in the active window by Stack Space.

## Windows Menu

The windows menu lists all the open profiler data windows.

## Profiler Function Reference

This is a reference for all Profiler functions mentioned in the text of this manual. The functions described in this chapter are:

- `ProfilerInit()`
- `ProfilerTerm()`
- `ProfilerSetStatus()`
- `ProfilerGetStatus()`
- `ProfilerGetDataSizes()`
- `ProfilerDump()`
- `ProfilerClear()`
- `ProfilerCreateThread()`
- `ProfilerDeleteThread()`
- `ProfilerSwitchToThread()`
- `ProfilerGetMainThreadRef()`

---

## Profiler API

The discussion of each function includes the following attributes:

- Description: A high-level description of the function
- Prototypes: The entire C/C++ prototypes for the function
- Remarks: Implementational or other notes about the function

### ProfilerInit()

Description	<p>ProfilerInit() prepares the profiler for use and turns the profiler on. The parameters tell the profiler how this collection run is going to operate, and how much memory to allocate. ProfilerInit() <i>must</i> be the first profiler call before you can call any other routine in the profiler API.</p>
Prototypes	<pre>typedef enum {     collectDetailed,     collectSummary } ProfilerCollectionMethod;  typedef enum {     ticksTimeBase,     timeMgrTimeBase,     microsecondsTimeBase,     PPCTimeBase,     bestTimeBase } ProfilerTimeBase;  pascal OSErr ProfilerInit(     ProfilerCollectionMethod method,     ProfilerTimeBase timeBase,     long numFunctions, short stackDepth);</pre>
Remarks	<p>ProfilerInit() attempts to allocate memory in the Process Manager heap (temporary memory). If it can't get the required memory in that heap, it tries in the current heap. This strategy minimizes the effect that the profiler has on the application's memory partition.</p> <p>ProfilerInit() can return memory manager errors such as memFullErr and paramErr. If paramErr is returned, it means that one of the constants passed in was out of range, or that ProfilerInit() has already been called.</p>

The `method` and `timeBase` parameters select the appropriate profiler options. The `numFunctions` parameter indicates the number of routines in the program for which the profiler should allocate buffer storage. If the profiler is operating in detailed mode, this number is internally increased (exponentially), because of the branching factors involved. The `stackDepth` parameter indicates how many routines deep the stack can get.

A call to `ProfilerInit()` must be followed by a matching call to `ProfilerTerm()`.

### **ProfilerTerm()**

**Description** `ProfilerTerm()` stops the profiler and deallocates the profiler's buffers. It calls `ProfilerDump()` to dump out any information that has not been dumped. `ProfilerTerm()` must be called at the end of a profile session.

**Prototypes** `void ProfilerTerm( void );`

**Remarks** If a program exits after calling `ProfilerInit()` without calling `ProfilerTerm()`, timers may be left running that could crash the machine.

### **ProfilerSetStatus()**

**Description** `ProfilerSetStatus()` lets you turn profiler recording on and off in the program. This makes it possible to profile specific sections of your code such as screen redraw or a calculation engine. The profiler output makes more sense if the profiler is turned on and off in the same routine, rather than in different routines.

**Prototypes** `pascal void ProfilerSetStatus( short on );`

**Remarks** This routine and `ProfilerGetStatus()` are the only profiler routines that may be called at interrupt time.

Pass 1 to turn recording on and 0 to turn recording off.

### **ProfilerGetStatus()**

**Description** `ProfilerGetStatus()` lets you query the profiler to determine if it is collecting profile information.

**Prototypes** `pascal short ProfilerGetStatus( void );`

---

Remarks	<p>This routine and <code>ProfilerSetStatus()</code> are the only profiler routines that may be called at interrupt time.</p> <p><code>ProfilerGetStatus()</code> returns a 1 if the profiler is currently recording, 0 if it is not.</p> <p><b>ProfilerGetDataSizes()</b></p>
Description	<p><code>ProfilerGetDataSizes()</code> lets you query the profiler for the current size of the data collected in the function and stack tables. This information can be used to tune the parameters passed to <code>ProfilerInit()</code>.</p>
Prototypes	<pre>pascal void ProfilerGetDataSizes(     long *functionSize,     long *stackSize);</pre>
Remarks	<p>If you have passed <code>collectDetailed</code> to <code>ProfilerInit()</code>, <code>ProfilerGetDataSizes()</code> returns the number of actual routines in the table, which may be larger than the value passed to <code>ProfilerInit()</code> in <code>numFunctions</code>. This is because the profiler multiplies <code>numFunctions</code> by 12 when it allocates the table. The multiplication is done so that you can easily switch between <code>collectDetailed</code> and <code>collectSummary</code> methods without changing the parameters.</p> <p><b>ProfilerDump()</b></p>
Description	<p><code>ProfilerDump()</code> dumps the current profile information without clearing it. The filename passed must be a Pascal string.</p>
Prototypes	<pre>pascal OSErr ProfilerDump( StringPtr filename );</pre>
Remarks	<p>This can be useful for dumping intermediate results on a long task. If the specified file already exists, a new file is created with an incrementing number appended to the filename. This allows the dump to be called inside a loop with a constant filename.</p> <p><code>ProfilerDump()</code> can return memory manager errors such as <code>memFullErr</code>, or file system errors.</p> <p><b>ProfilerClear()</b></p>
Description	<p><code>ProfilerClear()</code> clears any profile information from the buffers.</p>
Prototypes	<pre>pascal void ProfilerClear( void );</pre>

---

Remarks `ProfilerClear()` retains the settings of `collectionMethod` and `timeBase` that were set by `ProfilerInit()`. It does not deallocate the buffers.

## **ProfilerCreateThread()**

Description Use `ProfilerCreateThread()` to create a profiler thread structure to track the information in a new thread. You should call this routine whenever you create a new thread. This call creates the necessary profiler buffers and returns a reference used by the profiler. The `threadRef` provided by this routine must be stored with the thread and passed to `ProfilerSwitchToThread()` in the thread's `swapIn` proc.

Prototypes

```
typedef unsigned long ProfilerThreadRef;
pascal OSErr ProfilerCreateThread(
    short stackSize, long byteStackSize,
    ProfilerThreadRef *threadRef);
```

Remarks The `stackSize` parameter is the number of profiler frames to use. The `byteStackSize` parameter is the size in bytes of the stack that was created. This is done so that you can use these routines with any threads package, the profiler does not include any explicit calls to the Thread Manager.

This routine allocates memory in the current heap.

## **ProfilerDeleteThread()**

Description When a thread terminates, call `ProfilerDeleteThread()` to clean up the profiler thread tracking structures.

Prototypes

```
pascal void ProfilerDeleteThread(
    ProfilerThreadRef thread);
```

Remarks Returns `paramErr` if the `ProfilerThreadRef` passed in is not a `threadRef` returned from `ProfilerCreateThread()`.

## **ProfilerSwitchToThread()**

Description Call `ProfilerSwitchToThread()` from the `swapIn` proc of the thread, passing the `threadRef` returned from `ProfilerCreateThread()`.

Prototypes

```
pascal void ProfilerSwitchToThread(
    ProfilerThreadRef thread);
```



Remarks Returns paramErr if not a threadRef returned from ProfilerCreateThread().

### **ProfilerGetMainThreadRef()**

Description ProfilerGetMainThreadRef() returns the threadRef of the implicitly created main thread. threadRef is a pointer to an internal data structure.

Prototypes `pascal ProfilerThreadRef  
ProfilerGetMainThreadRef();`

Remarks Use this call to get the threadRef. Necessary to switch back to the main thread with ProfilerSwitchToThread().

---

**TIP** PowerPlant has been set up to make the profiler thread calls for you. Look at the threads classes for examples on how to use the profiler thread routines.

---



# Index

## Freescale Semiconductor, Inc.

---

### Symbols

#pragma directives, profiler 77

.arr File 53, 82

See also Generate Arrange File

\_\_copy\_vectors() 85

\_\_PROFILE\_ENTRY 58

\_\_PROFILE\_EXIT 58

\_\_sinit 50

~StProfileSection() 64

### A

abnormal termination 52

accuracy, Profiler 29

active profiler 16

API, including Profiler 25

Apple menu 81

asynchronous routines 51

### B

bestTimeBase 44, 63, 79

by +Children % of Total 84

by +Children command (Profiler) 84

by Average command (Profiler) 84

by Count command (Profiler) 83

by Function Name command (Profiler) 83

by Maximum command (Profiler) 84

by Minimum command (Profiler) 84

by Only % of Total command (Profiler) 84

by Only command (Profiler) 83

by Stack Space command (Profiler) 84

### C

cache hits, improving 53

call frequency, sorting functions by 53

Close command (Profiler) 81

code resources 49

CodeWarrior

described 8

installing 14

Collapse All command (Profiler) 34, 83

collectDetailed 63

collection method 29, 44

compiler directives 43, 77

### D

data

detailed 28

exporting 32

finding problems 36

object-based 28

sorting 31, 83

summary 28

viewing 27

data columns

contents 30

resizing 30

debugging

profiled code 58

depth-first traversal, sorting functions by 54

design problems, finding 34

Detailed command (Profiler) 83

detailed data, collecting 44

detailed view 33

finding design problems 34

directives

C/C++ 77

compiler 43

display accuracy, Profiler 29

dumping results 26

### E

early profilers 15

Edit menu 82

exceptions 52

exiting Profiler 26

ExitToShell() 52

Expand All command (Profiler) 34, 83

exporting data 32

### F

File Mappings panel 55

File menu 81

finding problems 36

function overflow 29

function-level profiling 24, 77

### G

- Generate Arrange File 54
  - menu item 82
- Generate Profiler Information 23
- Generate Weighted Arrange File 54
  - menu item 82

### I

- initialize Profiler 26
- installing CodeWarrior 14
- interface files 25
- interrupt tasks 51
- interrupt time
  - and profiler 87
- interrupt time, and profiler 86

### L

- Libraries
  - Profiler 39–41, 67
- Link Arrangement file
  - See .arr File
- link arrangement files 53
- longjmp() 52

### M

- MacApp profiling 51
- main thread
  - profiling 47
- memory usage 45, 78
- Menus
  - Apple menu 81
  - Edit 82
  - File 81
  - View 32, 82
  - Window 84
- microsecond 30
- microsecondsTimeBase 44, 80
- millisecond 30

### O

- Object command (Profiler) 83
- object performance 35
- object view 35
- Open command (Profiler) 27, 81
- overflow functions 29

- overflow stack 29
- Overhead 29

### P

- Page Setup command (Profiler) 82
- passive profiler 15
- PowerPC PEF Linker 53
- PPCTimeBase 44, 80
  - under Mac OS X 80
- preprocessor directives 43
  - C/C++ 77
- Print command (Profiler) 82
- Print Preview 82
- Print Setup 82
- printing profiler output 82
- Profiler
  - accuracy 29
  - active 16
  - components 20
  - defined 15
  - early 15
  - exiting 26
  - getting results 26
  - including API 25
  - initialize 26
  - libraries 39–41, 67
  - memory usage 45
  - passive 15
  - printing output 82
  - recursive calls 42
  - sampling 15
  - Using Debugger with 58
- Profiler Function Reference 84–89
  - ProfilerClear() 87
  - ProfilerCreateThread() 88
  - ProfilerDeleteThread() 88
  - ProfilerDump() 87
  - ProfilerGetDataSizes() 87
  - ProfilerGetMainThreadRef() 89
  - ProfilerGetStatus() 86
  - ProfilerInit() 85
  - ProfilerSetStatus() 86
  - ProfilerSwitchToThread() 88
  - ProfilerTerm() 86
- Profiler menus
  - See Menus
- Profiler window 28
  - data columns 29–31

header items 28  
 ProfilerClear() 87  
 ProfilerCreateThread() 47, 88  
 ProfilerDeleteThread() 47, 88  
 ProfilerDump() 52, 87  
 ProfilerGetDataSizes() 87  
 ProfilerGetMainThreadRef() 48, 89  
 ProfilerGetStatus() 86  
 ProfilerInit()  
     used in StProfileSection 63  
 ProfilerInit() 45, 78, 85  
     data buffer 29  
     problems with profiler 74  
     used in StProfileSection 63  
     warning 53, 59  
 ProfilerSetStatus() 86  
 ProfilerSwitchToThread() 47  
 ProfilerSwitchToThread() 88  
 ProfilerTerm() 52, 86  
     warning 53, 59  
 ProfilerThreadRef, defined 88  
 profiling  
     activating 23  
     asynchronous routines 51  
     by function 24, 77  
     code resources 49  
     exceptions 52  
     far code 40  
     inline functions 74  
     interface files 41  
     interrupt tasks 51  
     MacApp 51  
     main thread 47  
     near code 40  
     setjmp() 52  
     shared libraries 49  
     smart code 40  
     static constructors 50  
     strategy 61  
     threads 47  
     with RAM Doubler 62  
     with Speed Doubler 63  
 Profiling C++ Static Constructors 50  
 Profiling in PowerPlant 61-72  
 Profiling MacApp 51  
 Project Settings 23

**Q**

Quit command (Profiler) 82

**R**

RAM Doubler and profiling 62  
 recursive calls 42  
 requirements *See* system requirements  
 results  
     dumping 26  
     exporting 32  
     finding problems 36  
     opening 27  
     sorting 31, 83  
 RTC registers 74

**S**

sampling profiler 15  
 Save As command (Profiler) 26, 32, 82  
 Save Report As 82  
 saving results 26  
 setjmp() 52  
 shared libraries 49  
 sorting data 31, 83  
 Speed Doubler and profiling 63  
 stack overflow 29  
 stack space, finding problems 36  
 static constructors 50  
 StProfileSection 42, 63  
     data members 64  
     destructor 64  
 StProfileSection() constructor 63  
 Summary command (Profiler) 83  
 summary data 44  
 summary view 32  
 Suppress Warning Messages 57  
 system requirements 13

**T**

TB registers 74  
 Threads  
     Profiling 47  
     Viewing Results 48  
 tick, defined 30  
 ticksTimeBase 30, 44, 80  
 time hogs, finding 36

Timebase 29  
timebase 44, 79  
    defined 79  
timeMgrTimeBase 44, 80

## U

UnloadSeg () 59  
    in Profiling 63  
    problems with profiler 46, 51, 59, 76  
UPProfiler.cp 63  
Use “.arr” file 56

## V

view in profiler  
    detailed 33  
    object 35  
    summary 32  
View menu 32, 82

## W

what’s in this manual 8  
where to learn more 9  
Windows menu 84