CodeWarrior™ Development Studio IDE 5.5 User's Guide

Profiler Supplement

Revised 20030811

Freescale and the Freescale logo are registered trademarks of Freescale Corp. in the US and/or other countries. CodeWarrior is a trademark or registered trademark of Freescale Corp. in the US and/or other countries. All other tradenames and trademarks are the property of their respective owners.

Copyright © Freescale Corporation. 2006. ALL RIGHTS RESERVED.

No portion of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Freescale. Use of this document and related materials are governed by the license agreement that accompanied the product to which this manual pertains. This document may be printed for non-commercial personal use only in accordance with the aforementioned license agreement. If you do not have a copy of the license agreement, contact your Freescale representative or call 1-800-377-5416 (if outside the U.S., call +1-512-996-5300.

Freescale reserves the right to make changes to any product described or referred to in this document without further notice. Freescale makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Freescale 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 navigation or communication, emergency systems, or other applications with a similar degree of potential hazard**.

How to Contact Freescale

Corporate Headquarters	Freescale Corporation 7700 West Parmer Lane MD: PL56 Austin, TX 78729 U.S.A.
World Wide Web	`http://www.freescale.com`
Sales	Voice: 800-377-5416 Fax: 512-996-4910 Email: sales@freescale.com
Technical Support	Voice: 800-377-5416 Email: support@freescale.com

Table of Contents

1

Introduction

Welcome to the CodeWarrior Development Studio IDE 5.5 User's Guide Profiler Supplement. The 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 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 your CodeWarrior prodict, 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 installation. The release notes directory is also located at the top level of the CodeWarrior CD.

What's in This Manual

The CodeWarrior profiler is a core component of the CodeWarrior IDE and is launched when profiling completes on the target system or when you open a .mwp file. The profiler is actually a system consisting of three components: the profiler libraries, the calls you add to your code, and the Profiler window you use to view results. The profiler system works with C and C++ code. This manual often refers to the CodeWarrior profiler system as "CodeWarrior profiler," or simply as "the profiler."

Table 1.1 lists every chapter in this manual and describes the information contained in each.

Table 1.1 Contents of Chapters

Chapter	Description
Introduction	This chapter.
Getting Started	Overview and background information.
Using the Profiler	How-to guide to profiling your code.
Viewing Results	Guide to displaying profiling results.
Troubleshooting	Common problems and solutions.
Profiler Reference	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 documentation included with your product.

- 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 the 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 Profiler window you use to view results.
- Experienced with CodeWarrior Profiler
  - 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.
- 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.*

2

Getting Started

This chapter 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.

How would you like to know 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.

Finally, active profilers generate a large amount of raw information. This can lead to confusion and difficulty interpreting the results. The Profiler window 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.

3

Using the 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 Profiler window to view and analyze the profile results

Details of the API are discussed in Profiler Function Reference The Profiler window 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 Profiler window 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.

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 Profiler window 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 Function Reference 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:

- Add a profiler library to the project
- Turn on profiling
- Include the profiler API interface
- Initialize the profiler
- Dump the profile results
- Exit the profiler

In the steps that follow, we'll detail precisely what to do in both C and C++. 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.

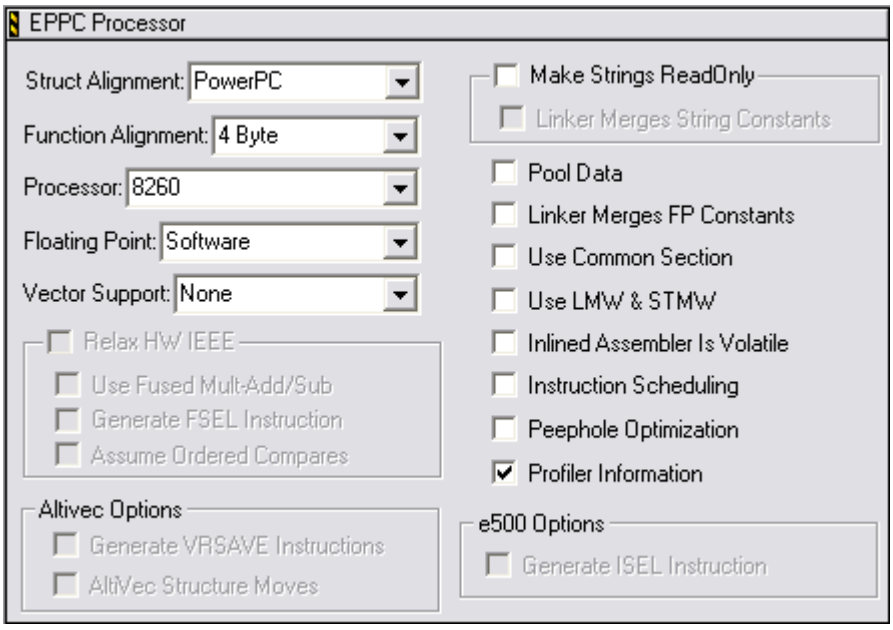
2. Turn on profiling

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

   a. 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 under the Code Generation option. Check the **Profiler Information** checkbox, as shown in Figure 3.1. With profiling on, the compiler generates all the code necessary so that every routine calls the profiler.

Figure 3.1 Processor Preferences Options for PowerPC



b. 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 your 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 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 Function Reference 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 profiler.

6. Exit the profiler

   When you are all through with the profiler, before exiting the program you should terminate the profiler by calling the correct profiler API function. On most platforms, 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. The IDE automatically opens this file in the Profiler window when the dump is complete. You can later re-open the file in the IDE to view the info again.

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.

4

Configuring the Profiler

This reference section discusses how to use the profiler libraries, APIs, and compiler options.

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 Libraries and Interface Files

You can find all of the profiler libraries and interface files in the Profiler 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. The libraries you use must match your settings in the ***Target*** settings panel.

The `profiler.h` file is the header file for the profiler API for C and C++. Include this file to make calls to conrol 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 abnormally terminated routines exited through the C++ exception handling model (try, throw, catch) or the ANSI C library `setjmp()` and `longjmp()` routines.

This section describes special cases you may encounter while profiling your code:

- Profiling Code with #pragma Statements

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" on page 16 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.

`#pragma profile on`	enables calls to the profiler in functions that are declared following the pragma
`#pragma profile off`	disables calls to the profiler in functions that are declared following the pragma
`#pragma profile reset`	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 4.1 shows the prototypes for `ProfilerInit()` for C/C++.

Table 4.1 ProfilerInit() Prototypes

C/C++	`long ProfilerInit(`
	`  ProfilerCollectionMethod method,`
	`  ProfilerTimeBase timeBase,`
	`  short numFunctions, short`
	`  stackDepth);`

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 4.2.

Table 4.2 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 one of the following values:

- ticksTimeBase
- microsecondsTimeBase
- timeMgrTimeBase
- PPCTimeBase
- win32TimeBase
- `bestTimeBase`

The `bestTimeBase` option automatically selects the most precise timing mechanism available on the computer running the profiled software. Not all of these values are supported on all target platforms. Refer to the Targeting Manual for your product to determine which timebases are available for use.

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 the profiled application's heap based on the method of collection, the number of routines, and the depth of the calling tree. On platforms where it is possible, the profiler will allocate memory outside of the application's heap, which helps reduce the profiler's effect on the application.

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.

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 4.3.

Table 4.3 ProfilerDump() Prototypes

C/C++	`long ProfilerDump(` `  unsigned char *filename);`

Calling ProfilerDump()

There is only one parameter: `char*`. The parameter points to a C-style string for filename. The IDE automatically adds a `.cwp` extension to the file name.

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 `exit()` is called. If you need to call `exit()` in the middle of your program and want the profiler output, call `ProfilerDump()`.

If you are using the profiler, you should always call `ProfilerTerm()` before `exit()`.

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

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 may 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 may 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.

CAUTION If you debug profiled code, you should not to kill the code from the debugger. If you have called `ProfilerInit()` you should call `ProfilerTerm()` on exit. If you do not do so, you may crash your system.

5

Viewing Results

This chapter discusses the ways you may view the data created by the CodeWarrior profiler.

In this chapter you will look at:

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

What It Does

The Profiler window displays profiler output for you to analyze the results of your program's execution. The profiler reads the 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

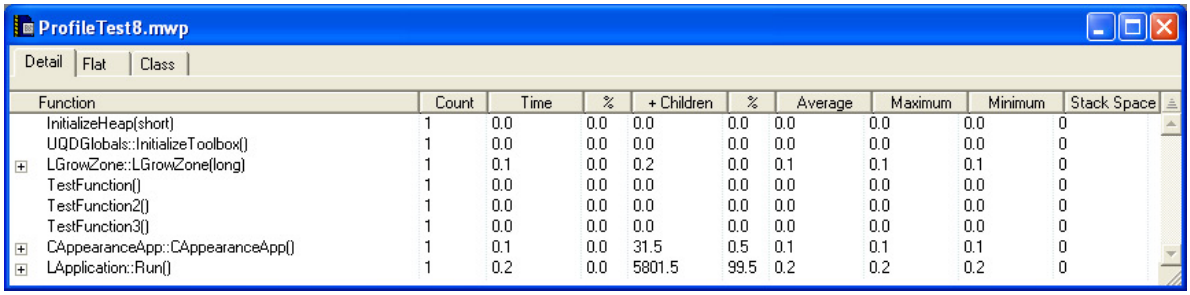
How It Works

You open profile data files exactly as you open files in any application. You can use the **Open** command from the **File** menu or drop the data file's icon on the CodeWarrior IDE. Whatever approach you take, when you open a file a window appears.

Profiler Window

The Profiler window allows you to view several elements of the profile data simultaneously, as shown in Figure 5.1.

Figure 5.1 Profiler Window



Profiler Window Data Columns

The profiler window contains 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 relative. Each time datum is reported to three decimal places. However, some time bases (most notably `ticksTimeBase`) are less precise. See Time and Timebases

Table 5.1 lists each of the columns in the profiler window (from left to right) and the information that column contains.

Table 5.1 Profile Window Data Columns

Column	Contents
Function name	Routine name. (The profiler unmangles C++ function names.)
Count	Number of times this routine was called.
Time	Time spent in this routine, not counting time in routines that this routine calls.
%	Percent of total time for the Time column.
+Children	Time spent in this routine and all the routines it calls.
%	Percent of total time for the +Children column.

Table 5.1 Profile Window Data Columns (*continued*)

Column	Contents
Average	Average time for each routine invocation: Time divided by the number of times the routine was called.
Maximum	Longest time for an invocation of the routine.
Minimum	Shortest time for an invocation of the routine.

Sorting Data

You can view the data sorted by the value in any column. To change the sort order, click the column title. The heading becomes highlighted and data is sorted by the value in that column. Use the arrow control to change the direction of the sort (ascending/ descending).

Multiple Windows

You can open any number of different profile windows simultaneously. This allows you to compare the results of different runs easily.

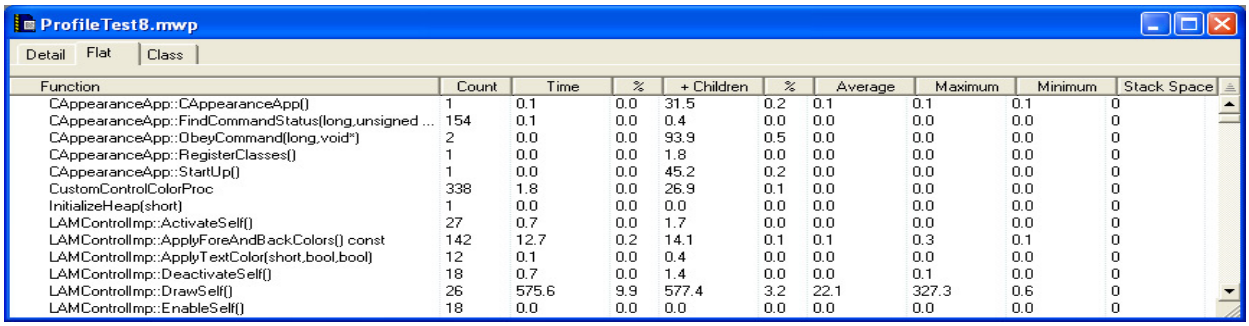
Window Views

In the tabs, you may choose to view the data in one of three ways: flat, detail, or class. Not all possibilities are available for all profiles.

Flat View

The flat 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 5.2 shows a flat view.

Figure 5.2 Flat View



Function	Count	Time	%	+ Children	%	Average	Maximum	Minimum	Stack Space
CAppearanceApp::CAppearanceApp()	1	0.1	0.0	31.5	0.2	0.1	0.1	0.1	0
CAppearanceApp::FindCommandStatus(long,unsigned ...	154	0.1	0.0	0.4	0.0	0.0	0.0	0.0	0
CAppearanceApp::ObeyCommand(long,void*)	2	0.0	0.0	93.9	0.5	0.0	0.0	0.0	0
CAppearanceApp::RegisterClasses()	1	0.0	0.0	1.8	0.0	0.0	0.0	0.0	0
CAppearanceApp::StartUp()	1	0.0	0.0	45.2	0.2	0.0	0.0	0.0	0
CustomControlColorProc	338	1.8	0.0	26.9	0.1	0.0	0.0	0.0	0
InitializeHeap(short)	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
LAMControlImp::ActivateSelf()	27	0.7	0.0	1.7	0.0	0.0	0.0	0.0	0
LAMControlImp::ApplyForeAndBackColors() const	142	12.7	0.2	14.1	0.1	0.1	0.3	0.1	0
LAMControlImp::ApplyTextColor(short,bool,bool)	12	0.1	0.0	0.4	0.0	0.0	0.0	0.0	0
LAMControlImp::DeactivateSelf()	18	0.7	0.0	1.4	0.0	0.0	0.1	0.0	0
LAMControlImp::DrawSelf()	26	575.6	9.9	577.4	3.2	22.1	327.3	0.6	0
LAMControlImp::EnableSelf()	18	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0

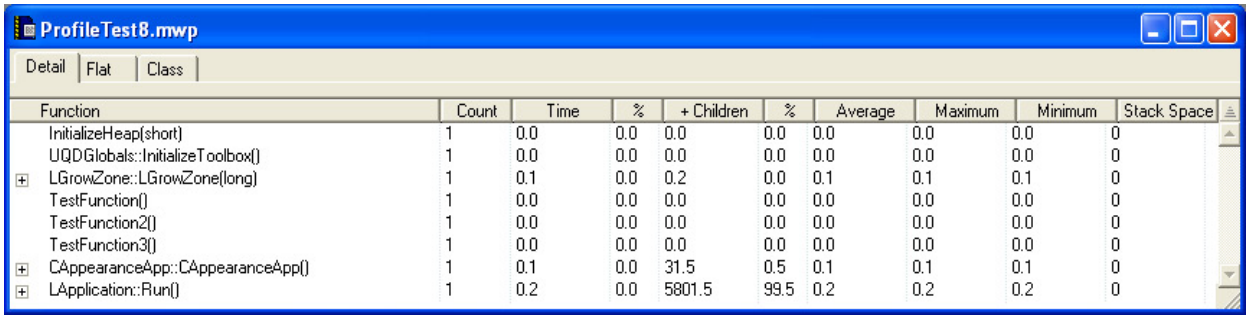
The flat view is particularly useful for comparing routines to see which take the longest time to execute. The flat view is also useful for finding a performance problem with a small routine that is called from many different places in the program. This view helps you look for the routines that make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

A flat view window can be displayed for any profile.

Detail View

The detail view displays routines according to the dynamic call tree as shown in Figure 5.3.

Figure 5.3 Detail View



Function	Count	Time	%	+ Children	%	Average	Maximum	Minimum	Stack Space
InitializeHeap(short)	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
UQDGlobals::InitializeToolbox()	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
⊞ LGrowZone::LGrowZone(long)	1	0.1	0.0	0.2	0.0	0.1	0.1	0.1	0
TestFunction()	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
TestFunction2()	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
TestFunction3()	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
⊞ CAppearanceApp::CAppearanceApp()	1	0.1	0.0	31.5	0.5	0.1	0.1	0.1	0
⊞ LApplication::Run()	1	0.2	0.0	5801.5	99.5	0.2	0.2	0.2	0

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 flat view for that purpose.

The detail view is useful for detecting design problems in code; it 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.

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

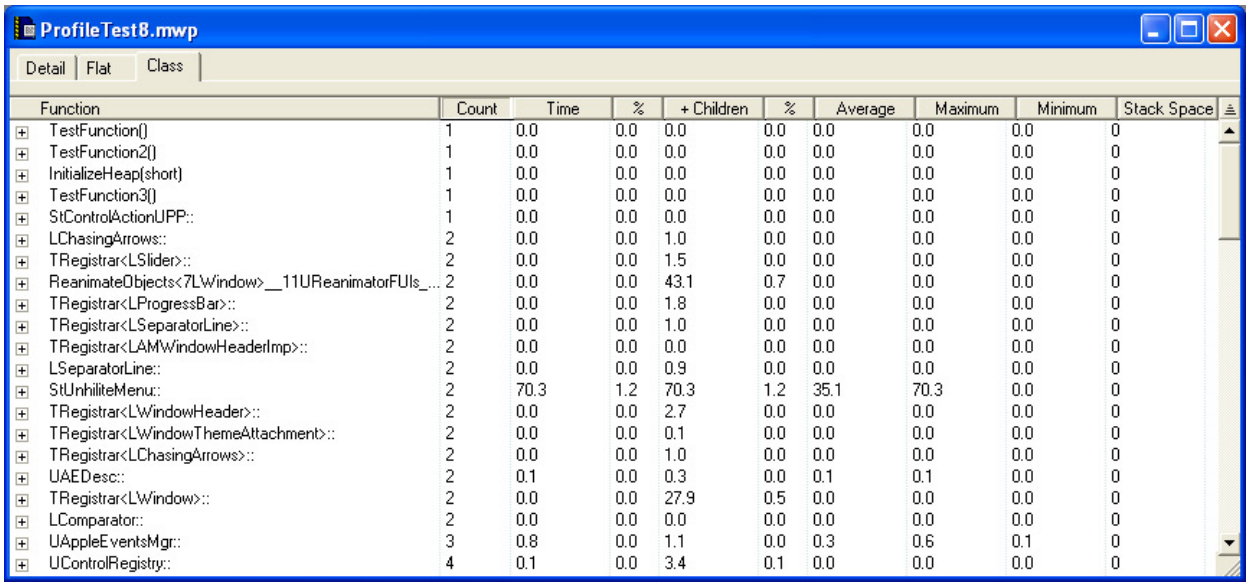
Class View

The class 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 detail view.

When sorting in class view, functions stay with their class, just like subsidiary functions in detail view stay in their hierarchical position. Figure 5.4 shows the methods sorted by count.

Class 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 flat view on a routine-by-routine basis, but the class 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 5.4 Class View



Function	Count	Time	%	+ Children	%	Average	Maximum	Minimum	Stack Space
TestFunction()	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
TestFunction2()	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
InitializeHeap(short)	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
TestFunction3()	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
StControlActionUPP::	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
LChasingArrows::	2	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0
TRegistrar<LSlider>::	2	0.0	0.0	1.5	0.0	0.0	0.0	0.0	0
ReanimateObjects<7LWindow>__11UReanimatorFUls_...	2	0.0	0.0	43.1	0.7	0.0	0.0	0.0	0
TRegistrar<LProgressBar>::	2	0.0	0.0	1.8	0.0	0.0	0.0	0.0	0
TRegistrar<LSeparatorLine>::	2	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0
TRegistrar<LAMWindowHeaderImp>::	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
LSeparatorLine::	2	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0
StUnhiliteMenu::	2	70.3	1.2	70.3	1.2	35.1	70.3	0.0	0
TRegistrar<LWindowHeader>::	2	0.0	0.0	2.7	0.0	0.0	0.0	0.0	0
TRegistrar<LWindowThemeAttachment>::	2	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0
TRegistrar<LChasingArrows>::	2	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0
UAEDesc::	2	0.1	0.0	0.3	0.0	0.1	0.1	0.0	0
TRegistrar<LWindow>::	2	0.0	0.0	27.9	0.5	0.0	0.0	0.0	0
LComparator::	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
UAppleEventsMgr::	3	0.8	0.0	1.1	0.0	0.3	0.6	0.1	0
UControlRegistry::	4	0.1	0.0	3.4	0.1	0.0	0.0	0.0	0

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 class 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 the 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 Time 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.

6

Troubleshooting

This chapter answers common questions about the profiler. So if you have a problem with the profiler, consult this chapter first. Other users may have encountered similar difficulties, and there may be a simple solution.

Profile Times Vary Between Runs

I'm getting different results (within 10%) in the 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 will encounter this problem.

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

Avoid this problem by increasing the number of times your function is called.

Problems while Profiling Inline Functions

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

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.

7

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
- Memory Usage—understanding memory usage
- Time and Timebases—the available time resolutions
- 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.

`#pragma profile on`	Enables calls to the profiler in functions that are declared following the pragma.
`#pragma profile off`	Disables calls to the profiler in functions that are declared following the pragma.
`#pragma profile reset`	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.

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()
{
#if __profile__    // is the profiler on?
                                                    if
(!ProfilerInit(collectDetailed, bestTimeBase, 20, 5))
                                                    {
#endif
                                                    test(15);
#if __profile__

ProfilerDump("Example.prof");
                                                    ProfilerTerm();
                                                    }
#endif
}
```

See also Routine-Level Profiling

Memory Usage

The profiler allocates two buffers in your program's heap 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()`.

When possible, the profiler will allocate its memory outside of your program's heap to reduce the impact of the profiler on your program. If this is not possible, the profiler's memory buffers will be allocated in your program's default heap. You must ensure that the heap is large enough to hold both your program's dynamically allocated data and the profiler's buffers.

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 40.

Time and Timebases

The `timeBase` may be one of the following values:

- ticksTimeBase
- microsecondsTimeBase
- timeMgrTimeBase
- PPCTimeBase
- win32TimeBase
- `bestTimeBase`

The `bestTimeBase` option automatically selects the most precise timing mechanism available on the computer running the profiled software. Not all of these values are supported on all target platforms. Refer to the Targeting Manual for your product to determine which timebases are available for use.

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

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()

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()

`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()` *must* be the first profiler call before you can call any other routine in the profiler API.

Prototypes

```
typedef enum {

collectDetailed,

collectSummary

} ProfilerCollectionMethod;

typedef enum {

bestTimeBase

} ProfilerTimeBase;

long ProfilerInit(

ProfilerCollectionMethod method,

ProfilerTimeBase bestTimeBase,

long numFunctions,short stackDepth);
```

Remarks

`ProfilerInit()` will allocate its memory outside of your program's heap to reduce the impact of the profiler on your program. If this is not possible, the profiler's memory buffers will be allocated in your program's default heap. You must ensure that the heap is large enough to hold both your program's dynamically allocated date and the profiler's buffers.

`ProfilerInit()` returns an error status that indicates whether or not the profiler was able to allocate it's memory buffers. If the return value is 0, then memory

allocation was successful. If a non-zero value is returned, then the allocation was not successful.

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()

`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.

```
void ProfilerTerm( void );
```

Remarks

If a program exits after calling `ProfilerInit()` you should call `ProfilerTerm()`. Failing to do so may lead to a crash on some platforms.

ProfilerSetStatus()

`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.

```
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()

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

```
short ProfilerGetStatus( void );
```

Remarks

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

`ProfilerGetStatus()` returns a 1 if the profiler is currently recording, 0 if it is not.

ProfilerGetDataSizes()

`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()`.

Prototypes

```
void ProfilerGetDataSizes(

long *functionSize,

long *stackSize);
```

Remarks

If you have passed `collectDetailed` to `ProfilerInit()`, `ProfilerGetDataSizes()` returns the number of actual routines in the table, which may be larger than the value passed to `ProfilerInit()` in `numFunctions`. This is because the profiler multiplies `numFunctions` by 12 when it allocates the table. The multiplication is done so that you can easily switch between `collectDetailed` and `collectSummary` methods without changing the parameters.

ProfilerDump()

`ProfilerDump()` dumps the current profile information without clearing it.

```
long ProfilerDump( char* filename );
```

Remarks

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.

A non-zero value from `ProfilerDump()` indicates that an error has occured.

ProfilerClear()

`ProfilerClear()` clears any profile information from the buffers.

```
void ProfilerClear( void );
```

Remarks

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

Index

R

S

T

V

W