CodeScape

Statistical Profiler Guide

Legal Notice

IMPORTANT

Revision History

Release Version 2.3 January 2000

Release Version 2.2.0 August 1999

Cross Products Ltd
23 The Calls, Leeds, West Yorkshire, LS2 7EH
telephone: +44 113 242 9814
facsimile: +44 113 242 6163
www.crossproducts.co.uk
email sales: enquiry@crossproducts.co.uk
email support: support@crossproducts.co.uk

Contents

About this guide

This guide is divided into four sections:

What is profiling? provides a brief description of the principles and objectives behind statistical profiling.

Statistical Profiler Tutorial takes you in simple steps through setting up the Profiler, profiling a simple program, and displaying and interpreting the results. By completing the tutorial, which takes about half an hour, you will be familiar with all the Statistical Profiler's features.

Appendix A explains the features on the Profilers's shortcut menu.

Appendix B gives a detailed description of the format of a Profiler data file (.prf).

NOTE: The example files referred to in the tutorial are included on the release CD in Tutorials\Profiler\Exam1 and ...\Exam2.

What is profiling?

Statistical profiling

The Statistical Profiler included with CodeScape is an analysis tool for examining the run-time behavior of programs written for Hitachi SuperH^{TM} microprocessors. It shows where your program spends its time so you can identify inefficient sections of code.

When your program executes, the Profiler takes regular samples of the processor's program counter (PC) at a configurable rate from 1Hz to 1kHz. From the address returned with each PC sample it can resolve which function your program is in at that moment in time.

Over a period of execution time, a profile of these samples is built-up which shows the approximate percentage of time spent in each function, source line, and instruction of your code. These values are used to identify hot spots where your program spends disproportionate amounts of time.

Statistical profiling is passive which means it does not modify your code to gather performance data. Therefore your program's execution time is not noticeably affected when the Profiler is running.

Function trace profiling

Function trace profiling lets you generate function and trace profiles which provide more detail about how specific functions are called. The method employed in CodeScape to do this depends on the model of Hitachi SuperH^{TM} microprocessor you are debugging. Contact enquiries@crossproducts.co.uk for more information about function trace profiling with CodeScape.

Statistical Profiler Tutorial

About this tutorial

This tutorial assumes you are running CodeScape on a supported Hitachi SuperH^{TM} target microprocessor. It also assumes that you know how to use CodeScape and are familiar with its regular functions and regions.

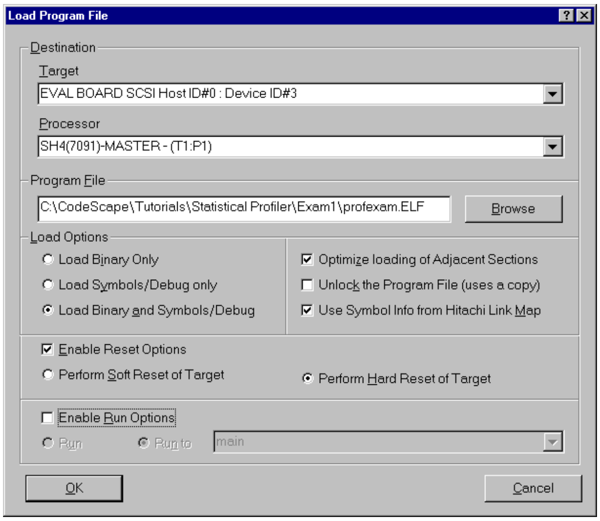
The example files used in this tutorial can be found in the CodeScape installation on your development PC under Tutorials\Profiler\Exam1 and Exam2. If you did not select the Tutorials option during installation, the tutorial files can be copied from the CodeScape CD.

Profiling example 1

Loading the example program file

1. Start CodeScape.
2. In the Target window, right-click and select Load Program File...
3. Browse to the file Tutorials\Profiler\Exam1\profexam.elf.
4. Make the settings in the Load Program File dialog box as shown in *Figure 1*.

Figure 1: Load Program File options


Load Program File

Destination
Target: EVAL BOARD SCSI Host ID#0 : Device ID#3
Processor: SH4(7091)-MASTER - (T1:P1)

Program File: C:\CodeScape\Tutorials\Statistical Profiler\Exam1\profexam.ELF

Load Options
- ○ Load Binary Only
- ○ Load Symbols/Debug only
- ● Load Binary and Symbols/Debug
- ☑ Optimize loading of Adjacent Sections
- ☐ Unlock the Program File (uses a copy)
- ☑ Use Symbol Info from Hitachi Link Map

☑ Enable Reset Options
- ○ Perform Soft Reset of Target
- ● Perform Hard Reset of Target

☐ Enable Run Options
- ○ Run
- ● Run to: main

5. Click OK.
   A progress box is displayed briefly and profexam.elf is downloaded onto the target.

Setting up the Profiler

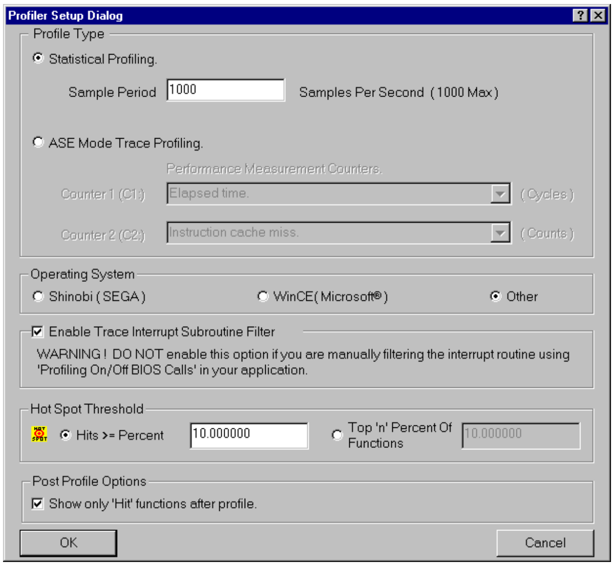
1. From the Tools menu select Profiler.
   The Profiler Setup dialog box appears in CodeScape as shown in Figure 2.

The steps below describe what settings to make in the Profiler Setup dialog box.

2. Select *Statistical Profiling.*
3. In the *Sample Period* text box, set the sample rate to 1000Hz.
   This is the rate at which the Profiler samples the target's PC. There is no fixed rule to determine the optimum sample rate. The higher the sample rate, and the longer the duration of the profile, the more accurate the result will be. Normally the maximum sample rate of 1000Hz will not noticeably slow down the execution time of your program so we recommend that you always use this setting.

NOTE: On some SH2 and SH3 processors running at slower clock speeds, the sample rate may have a greater effect on execution time but it should still be negligible.

Figure 2: Profiler Setup dialog



Profiler Setup Dialog

Profile Type

[✓ Statistical Profiling.]

Sample Period 1000 Samples Per Second (1000 Max)

ASE Mode Trace Profiling.

Performance Measurement Counters.

Counter 1 (C1:) Elapsed time. (Cycles)

Counter 2 (C2:) Instruction cache miss. (Counts)

Operating System

Shinobi (SEGA) WinCE (Microsoft®) [✓ Other]

[✓ Enable Trace Interrupt Subroutine Filter]

WARNING ! DO NOT enable this option if you are manually filtering the interrupt routine using 'Profiling On/Off BIOS Calls' in your application.

Hot Spot Threshold

[✓ Hits >= Percent] 10.000000 Top 'n' Percent Of Functions 10.000000

Post Profile Options

[✓ Show only 'Hit' functions after profile.]

4. In the *Hot Spot Threshold* box click *Hits >= Percent*. In the text box enter 10%. This specifies the threshold above which functions are labeled with a hot spot in the Profiler results window.
   - *Hits >= Percent* labels all functions that exceed the specified percentage of the total number of hits counted during the profile.
   - *Top 'n' Percent of Functions* sorts the functions according to the number of times they were hit during the profile, the top 'n' percent of which are then labeled as hot spots.

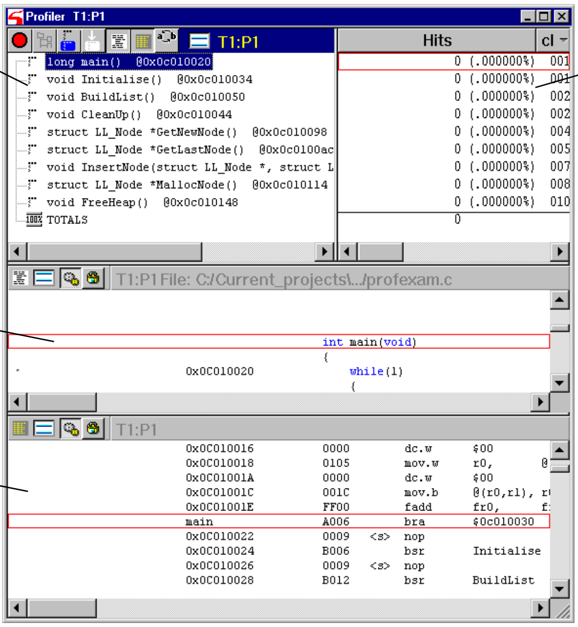
   The value entered for this exercise of 10% is a rather crude value because the example program has only a few functions. For larger, practical applications you would make use of the six decimal places provided to set up lower hot spot thresholds with greater resolution.
5. Select *Show only 'Hit' functions after profile.*
   With this selected the results of the Profile will show only those functions that were hit during the profile. All other functions are not displayed. This is a useful feature for profiling large programs.
6. Click OK.
7. In the Profiler window click and to display Source and Disassembly regions as shown in *Figure 3*.

Figure 3: The Profiler window before performing a profile

List of functions in the example program. Only those functions whose names can be resolved from debug information are shown in the list.

Source region.

Disassembly region.



```
long main()  @0x0c010020                          0 (.000000%)  001
void Initialise()  @0x0c010034                    0 (.000000%)  001
void BuildList()  @0x0c010050                     0 (.000000%)  002
void CleanUp()  @0x0c010044                       0 (.000000%)  002
struct LL_Node *GetNewNode()  @0x0c010098         0 (.000000%)  004
struct LL_Node *GetLastNode()  @0x0c0100ac        0 (.000000%)  005
void InsertNode(struct LL_Node *, struct L        0 (.000000%)  007
struct LL_Node *MallocNode()  @0x0c010114         0 (.000000%)  008
void FreeHeap()  @0x0c010148                      0 (.000000%)  010
TOTALS                                            0
```

T1:P1 File: C:/Current_projects\.../profexam.c

```
                                    int main(void)
                                    {
            0x0C010020                  while(1)
                                        {
```

T1:P1

```
            0x0C010016      0000        dc.w      $00
            0x0C010018      0105        mov.w     r0,       @
            0x0C01001A      0000        dc.w      $00
            0x0C01001C      001C        mov.b     @(r0,r1), r
            0x0C01001E      FF00        fadd      fr0,      f
            main            A006        bra       $0c010030
            0x0C010022      0009   <s>  nop
            0x0C010024      B006        bsr       Initialise
            0x0C010026      0009   <s>  nop
            0x0C010028      B012        bsr       BuildList
```

Hits counter, showing the number of hits on each function. This is also expressed as a percentage of the total number of hits on all the functions in the profile.

The “cl” column shows the cache line number for each function.

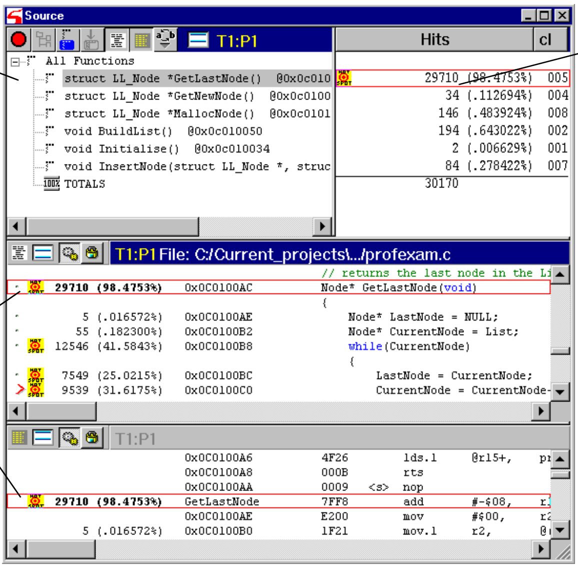
Starting the profiler and running the program

1. Click ● (red) to start the Profiler.
   The button is green when the Profiler is running and red when the Profiler is stopped.

2. Press F9 to run the program. Let the program run for about 30 seconds and press F9 again to stop it.
   The Hits counter updates while the program is running.
   The length of time that you profile the program for depends on the size of the program you are debugging. The longer you allow the program to run, the greater the number of samples taken by the Profiler, and therefore the greater the accuracy of the result.

3. Click ● (green) to stop the Profiler.
   The Profiler takes a few seconds to resolve the data, after which you see the final results in the Profiler window as shown in *Figure 4*.

Figure 4: The Profiler results display



```
All Functions
  struct LL_Node *GetLastNode()  @0x0c010      29710 (98.4753%)  005
  struct LL_Node *GetNewNode()  @0x0c0100         34 (.112694%)  004
  struct LL_Node *MallocNode()  @0x0c0101        146 (.483924%)  008
  void BuildList()  @0x0c010050                  194 (.643022%)  002
  void Initialise()  @0x0c010034                   2 (.006629%)  001
  void InsertNode(struct LL_Node *, struc         84 (.278422%)  007
  TOTALS                                       30170
```

```
T1:P1 File: C:/Current_projects\../profexam.c
                                      // returns the last node in the Li
 29710 (98.4753%)   0x0C0100AC        Node* GetLastNode(void)
                                      {
     5 (.016572%)   0x0C0100AE            Node* LastNode = NULL;
    55 (.182300%)   0x0C0100B2            Node* CurrentNode = List;
 12546 (41.5843%)   0x0C0100B8            while(CurrentNode)
                                          {
  7549 (25.0215%)   0x0C0100BC                LastNode = CurrentNode;
  9539 (31.6175%)   0x0C0100C0                CurrentNode = CurrentNode-
```

```
T1:P1
                    0x0C0100A6   4F26        lds.l   @r15+,   pr
                    0x0C0100A8   000B        rts
                    0x0C0100AA   0009   <s>  nop
 29710 (98.4753%)   GetLastNode  7FF8        add     #-$08,   r
                    0x0C0100AE   E200        mov     #$00,    r2
     5 (.016572%)   0x0C0100B0   1F21        mov.l   r2,      @
```

List of the functions hit during the profile. Functions that are not hit are not shown.

Column showing the number of hits on each function. One hot spot is shown in this example.

Source and Disassembly regions showing the same hot spot.

4. Click on HOT SPOT in the Hits column.
   The cursors in the Source and Disassembly regions automatically move to locate the hot spot in the code. The point at which the icon is shown indicates a hot spot where the number of hits on that function meets the threshold requirements specified earlier.

NOTE: *Sometimes the Profiler cannot resolve the function name from the address returned by the PC sample, usually because of insufficient debug data. In this case the hits are scored against "Other functions" in the function list. The number hits on "Other functions" can include hits from more than one function.*

Analyzing the profile data

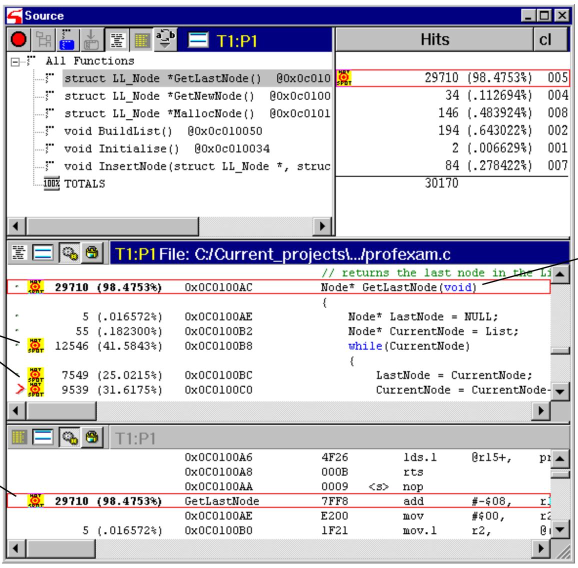
This is a simple program that creates a single-ended linked list of ascending numbers from 1 to 1000, clears the list, and then starts again.

The hot spot in the Hits column shows that the program spent 98.5% of the time during the profile sample in the function GetLastNode. In this program, spending 98% of the execution time in just one function is extremely inefficient.

The Hits column does not show the actual number of hits on each function during the profile, instead, statistical profiling gives the number of hits detected by a periodic sample. Therefore the number of hits detected during statistical profiling is an indication of the time spent in each function rather than the actual number of hits on each function.

In the function GetLastNode, the program locates the last entry in the list by searching from the start of the list until it reaches the last node. The problem occurs in a particular loop in GetLastNode, this is highlighted by another three hot spots beneath GetLastNode which are shown in the Source region.

Figure 5: Analyzing the Profiler results



```
Source
T1:P1                                                    Hits                cl
All Functions
  struct LL_Node *GetLastNode()  @0x0c010            29710 (98.4753%)  005
  struct LL_Node *GetNewNode()  @0x0c0100               34 (.112694%)  004
  struct LL_Node *MallocNode()  @0x0c0101              146 (.483924%)  008
  void BuildList()  @0x0c010050                        194 (.643022%)  002
  void Initialise()  @0x0c010034                         2 (.006629%)  001
  void InsertNode(struct LL_Node *, struc               84 (.278422%)  007
  TOTALS                                             30170
```

```
T1:P1 File: C:/Current_projects\.../profexam.c
                                             // returns the last node in the Li
  29710 (98.4753%)   0x0C0100AC              Node* GetLastNode(void)
                                             {
      5 (.016572%)   0x0C0100AE                  Node* LastNode = NULL;
     55 (.182300%)   0x0C0100B2                  Node* CurrentNode = List;
  12546 (41.5843%)   0x0C0100B8                  while(CurrentNode)
                                                 {
   7549 (25.0215%)   0x0C0100BC                      LastNode = CurrentNode;
   9539 (31.6175%)   0x0C0100C0                      CurrentNode = CurrentNode-
```

```
T1:P1
                     0x0C0100A6      4F26         lds.l    @r15+,    pr
                     0x0C0100A8      000B         rts
                     0x0C0100AA      0009    <s>  nop
  29710 (98.4753%)   GetLastNode     7FF8         add      #-$08,    r1
                     0x0C0100AE      E200         mov      #$00,     r2
      5 (.016572%)   0x0C0100B0      1F21         mov.l    r2,       @(
```

The loop shown in Source where the program spends a disproportionate amount of time.

GetLastNode shown at instruction level in the Disassembly region.

The total number of hits (bold) shown next to a function is the sum of hits (non-bold) on all the source lines belonging to that function.

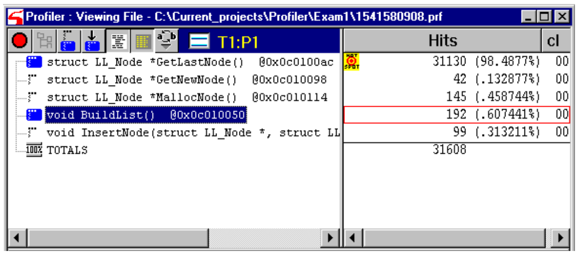
The number in bold next to GetLastNode in the Source region shows the total number of hits on that function, and the numbers beneath show how many times each line in that function was hit during the profile.

You can see the hits broken down into even more detail at instruction level in the Disassembly region.

The Profiler's results show that rewriting GetLastNode so that it does not search from the start of the list each time will remove the inefficiency and speed up the program. You will see this later in Example 2.

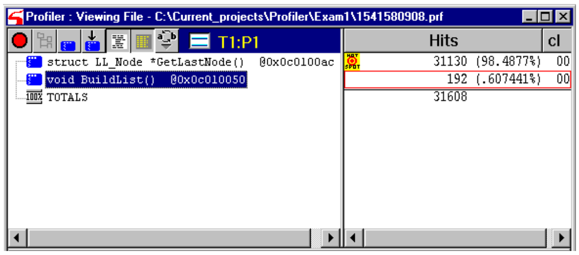
Tagging functions and displaying the results using different views

1. In the function list, double-click on the functions GetLastNode and BuildList.
   The functions are tagged with a blue rectangle.



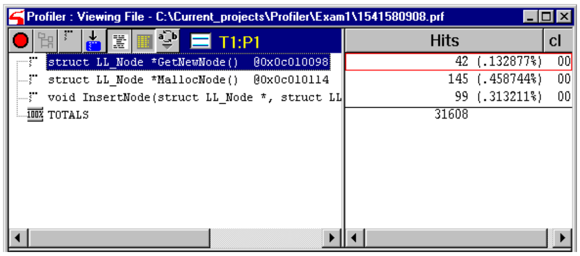
```
Profiler : Viewing File - C:\Current_projects\Profiler\Exam1\1541580908.prf
T1:P1                                                   Hits                  cl
struct LL_Node *GetLastNode()   @0x0c0100ac       31130 (98.4877%)   00
struct LL_Node *GetNewNode()   @0x0c010098           42 (.132877%)   00
struct LL_Node *MallocNode()   @0x0c010114          145 (.458744%)   00
void BuildList()  @0x0c010050                       192 (.607441%)   00
void InsertNode(struct LL_Node *, struct LL          99 (.313211%)   00
TOTALS                                            31608
```

2. Click .
   The button changes to and the view changes to show only the tagged functions.



```
Profiler : Viewing File - C:\Current_projects\Profiler\Exam1\1541580908.prf
T1:P1                                                   Hits                  cl
struct LL_Node *GetLastNode()   @0x0c0100ac       31130 (98.4877%)   00
void BuildList()  @0x0c010050                       192 (.607441%)   00
TOTALS                                            31608
```

3. Click .
   The button changes to and the view changes to show only the functions that are not tagged.



```
Profiler : Viewing File - C:\Current_projects\Profiler\Exam1\1541580908.prf
T1:P1                                                   Hits                  cl
struct LL_Node *GetNewNode()   @0x0c010098           42 (.132877%)   00
struct LL_Node *MallocNode()   @0x0c010114          145 (.458744%)   00
void InsertNode(struct LL_Node *, struct LL          99 (.313211%)   00
TOTALS                                            31608
```

4. Click .
   The button changes to and the view reverts to show all functions.

Where you have large function lists in your profile results:

5. Click to go to the next tagged function in the list.
6. Click to sort the functions alphabetically, either ascending or descending.
7. In the Hits column, click *Hits* or *cl* to sort the list either by Hits; ascending or descending, or by cache line number; ascending or descending.

Using the search facility

NOTE: This feature is useful for large function lists. You can search for strings, whole words, or parts of words.

1. Click on GetLastNode in the function list.
2. Type “bu”.
   The cursor stops at BuildList.
   The search begins from the cursor and looks for exact matches first, then the nearest matches in descending order.
3. To continue the search press F3 or Enter.

Renaming a function

1. Click on GetLastNode in the function list.
2. Right-click, click Rename Function...
   The Rename Function dialog box appears.
3. In the dialog box type “HotSpot” and click OK.
   The function is renamed HotSpot in the Profiler’s function list.

NOTE: This does not change the function’s name in the source code. Original source files remain unchanged, only the Profiler’s view changes.

Saving the profile data

As profile data

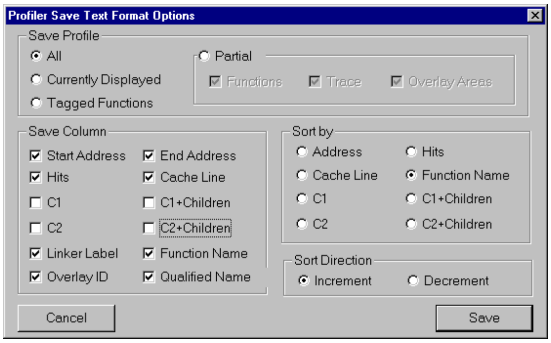
1. Right-click in the Profiler window, point to File, Save Profile and click Save Profile (.PRF)...
2. In the dialog box browse to the location where you want to save the profile data, name the file and click Save.
   You can close the session and return to it later and reload the profile results. Only the profile data and Profiler Breakpoints are saved, the Profiler display settings and hot spot thresholds are lost.

NOTE: The format of .PRF files is given in detail in Appendix B.

As text

1. Right-click in the Profiler window, point to File, Save Profile and click Save Profile (.TXT)...
2. Fill in the options that you want to save in the text file and click Save.
   The options are described in *Table 1*.

Figure 6: Save Profile Text Option



Save Profile
- [✓ All] Currently Displayed Tagged Functions
- Partial: [✓ Functions] [✓ Trace] [✓ Overlay Areas]

Save Column
- [✓ Start Address] [✓ End Address]
- [✓ Hits] [✓ Cache Line]
- C1 C1+Children
- C2 C2+Children
- [✓ Linker Label] [✓ Function Name]
- [✓ Overlay ID] [✓ Qualified Name]

Sort by
- Address Hits
- Cache Line [✓ Function Name]
- C1 C1+Children
- C2 C2+Children

Sort Direction
- [✓ Increment] Decrement

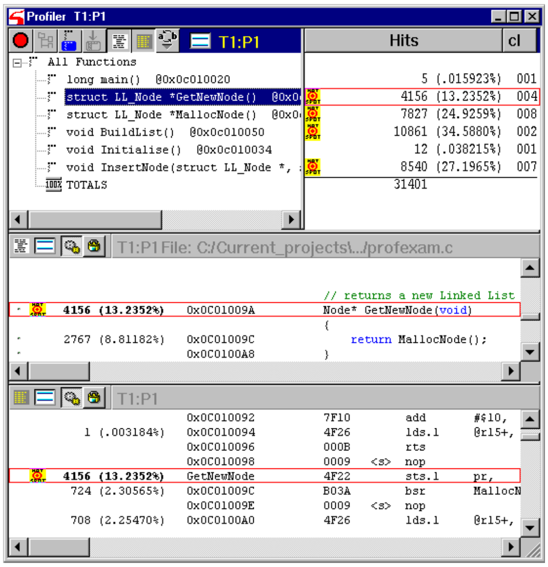
Table 1: Text format save options

Save Profile	
All	Saves the profile data for all the functions listed in the Profiler results.
Currently Displayed	Saves the profile data for only the functions visible in the function list in the Profiler window.
Tagged Functions	Saves the profile data for only the functions that are tagged in the function list.
Partial	
Functions	Saves data associated only with functions.
Trace	Saves data associated only with Trace profiling.
Overlay Areas	Saves additional overlay information if overlays are in use.
Save Column	
Start Address	Saves the start address of each function, this is the address shown in the Profiler function list.
End Address	Saves the end address of each function, this is not shown in the Profiler function list.
Hits/Cache Line	Saves the data in the Hits and Cache Line columns
Linker Label/Function Name	Saves each function’s name and linker label.
Overlay ID	Saves each function’s overlay ID if overlays are in use.
Qualified Name	Saves the function’s qualified name (function name + file name + class name) if the function is not unique.
Sort by	
Address, Hits, Cache Line or Function Name	Sorts the profile data by Address, Hits, Cache Line or Function Name as selected.
Sort Direction	
Increment/Decrement	Direction in which the columns are to be sorted in the text file.

Profiling example 2

1. In the Target window, right-click and select Load Program File...
2. Browse to the file Tutorials\Profiler\Exam2\profexam.elf and click OK. Keep the Load Program File options as specified for example 1.
   A progress box is displayed briefly and profexam.elf is loaded onto the target.
3. Click ● (red) to start the Profiler.
4. Press F9 to run the program. Let the program run for about 30 seconds and press F9 again to stop it.
5. Click ● (green) to stop the Profiler.
   The Profiler resolves the data and shows the final results:

Figure 7: Profile results for example 2



```
All Functions
  long main()  @0x0c010020                       5 (.015923%)   001
  struct LL_Node *GetNewNode()  @0x0          4156 (13.2352%)   004
  struct LL_Node *MallocNode()  @0x0          7827 (24.9259%)   008
  void BuildList()  @0x0c010050              10861 (34.5880%)   002
  void Initialise()  @0x0c010034                12 (.038215%)   001
  void InsertNode(struct LL_Node *, :         8540 (27.1965%)   007
  TOTALS                                     31401
```

T1:P1 File: C:/Current_projects\.../profexam.c

```
                                         // returns a new Linked List
4156 (13.2352%)   0x0C01009A            Node* GetNewNode(void)
                                        {
2767 (8.81182%)   0x0C01009C                return MallocNode();
                  0x0C0100A8            }
```

T1:P1

```
                  0x0C010092    7F10        add      #$10,
   1 (.003184%)   0x0C010094    4F26        lds.l    @r15+,
                  0x0C010096    000B        rts
                  0x0C010098    0009   <s>  nop
4156 (13.2352%)   GetNewNode    4F22        sts.l    pr,
 724 (2.30565%)   0x0C01009C    B03A        bsr      MallocN
                  0x0C01009E    0009   <s>  nop
 708 (2.25470%)   0x0C0100A0    4F26        lds.l    @r15+,
```

Analyzing the profile data

In example 2, the program has been rewritten to use a local pointer to remember the last inserted node. This means that GetLastNode does not appear in the profile results because the function is no longer used and has not been hit during the profile.

Instead, you can see from the hot spots, that most of the work in this simple program is shared fairly evenly between four functions which is a much more efficient use of execution time.

Profiling a section of code using Profiler breakpoints

This feature is used to isolate a section of code for profiling. Due to the nature of statistical profiling it is usually preferable to profile the complete program to get a picture of how the program performs as a whole, so you would not normally use Profiler breakpoints.

There are two types of Profiler breakpoints, start and stop, these are different to normal breakpoints. When a start breakpoint occurs, an instruction is sent to the Profiler to start profiling. The Profiler runs until either a stop breakpoint occurs or until it is stopped manually. Unlike other breakpoints, on hitting a Profiler breakpoint, the execution of the code does not stop, instead the program continues to run. You can set other breakpoints in CodeScape that trigger different actions during profiling.

- You can only set one Profiler start breakpoint, but you can set multiple Profiler stop breakpoints.
- If the start breakpoint is at the beginning of a loop then you should set at least one stop breakpoint at the end of the loop otherwise a recursion can occur in the Profiler function call tree.
- Program execution does not stop when a Profiler breakpoint occurs.
- It takes a finite amount of time to process a Profiler breakpoint, therefore a frequently hit breakpoint increases the time it takes to profile the program but does not effect the processor time (i.e.. the number of clock cycles).

NOTE: *You can set up breakpoints in CodeScape that will trigger the Profiler to start. You insert the breakpoints in your program in the normal way using CodeScape Source and Disassembly regions and then configure the Trigger Actions dialog box to start or stop the Profiler upon hitting the breakpoints. If you use this method, you must remember to enable the Profiler before running the program (stop/start button is green when the Profiler is enabled)*

Loading the program file

In the Target region, right-click and load profexam.elf from the Exam2 directory.

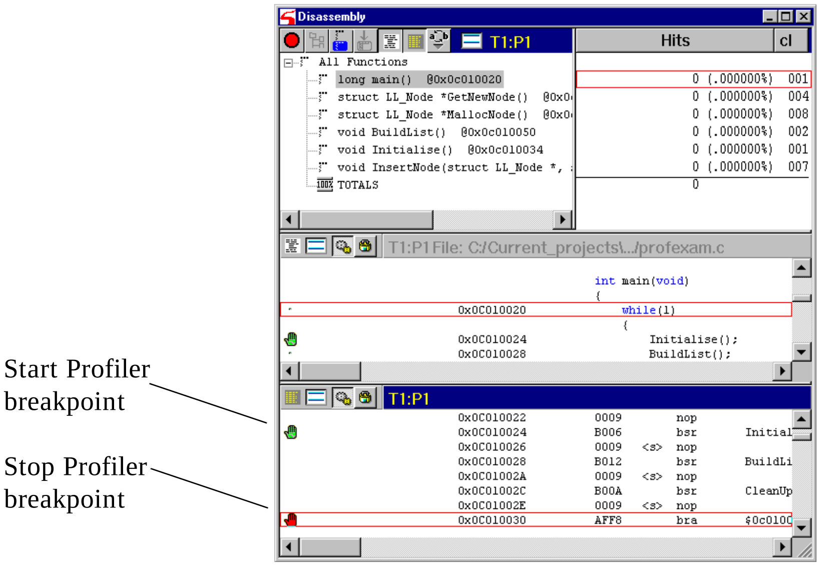
If you already have example 2 loaded from the previous exercise press CTRL+ALT+R. This is the default keyboard shortcut to restart the target, reload the executable code and reset the Profiler ready for profiling.

Setting the Profiler breakpoints

1. In the function list click on *main*.
   The cursor in the Disassembly region stops on *main*.

2. Click on *Main* in the Disassembly region.
3. Press the down arrow key twice until the cursor stops on *Initialise*.
4. Press F5 to insert a start breakpoint.
   A green hand appears at the insertion point to indicate a start breakpoint.
5. Press the down arrow key six times to scroll down to the *bra* instruction at 0x0C010030 in the Disassembly region.
6. Press F5 to insert a stop breakpoint.
   A red hand appears at the insertion point to indicate a stop breakpoint. The Profiler automatically decides whether the breakpoint is a start or a stop based on your previous action.

Figure 8: Profiler start and stop breakpoints


Start Profiler breakpoint
Stop Profiler breakpoint
Disassembly
T1:P1
Hits
cl
T1:P1 File: C:/Current_projects\.../profexam.c
T1:P1

```
All Functions
   long main()  @0x0c010020                      0 (.000000%)  001
   struct LL_Node *GetNewNode()  @0x0            0 (.000000%)  004
   struct LL_Node *MallocNode()  @0x0            0 (.000000%)  008
   void BuildList()  @0x0c010050                 0 (.000000%)  002
   void Initialise()  @0x0c010034                0 (.000000%)  001
   void InsertNode(struct LL_Node *, :           0 (.000000%)  007
   TOTALS                                        0
```

```
                                  int main(void)
                                  {
      0x0C010020                      while(1)
                                      {
      0x0C010024                          Initialise();
      0x0C010028                          BuildList();
```

```
      0x0C010022      0009           nop
      0x0C010024      B006           bsr      Initial
      0x0C010026      0009    <s>    nop
      0x0C010028      B012           bsr      BuildLi
      0x0C01002A      0009    <s>    nop
      0x0C01002C      B00A           bsr      CleanUp
      0x0C01002E      0009    <s>    nop
      0x0C010030      AFF8           bra      $0c010
```

You can see that only one breakpoint is visible in the Source region. This is because the loop that you have isolated to profile begins and ends on the same instruction in the source code.

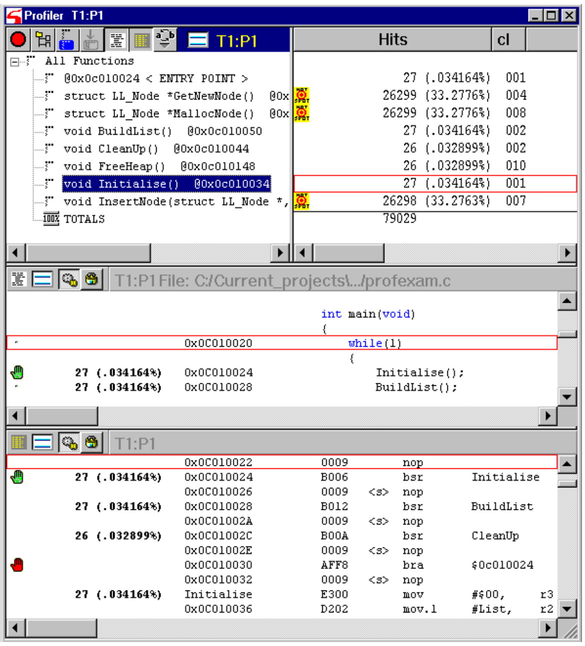
In some circumstances the stop breakpoint can appear before the start breakpoint in the Source region. This is due to the generated executable code being different from the source code as a result of compiler optimizations. You can always see exactly where the breakpoints occur by looking at the Disassembly region.

NOTE:	*When setting Profiler breakpoints using the Source region, always check the Disassembly region because the Source addresses may not reflect the actual executable code.*

Starting the Profiler and running the program

1. Click ● (red) to start the Profiler.
2. Press F9 to run the program. Let the program run for about 30 seconds and press F9 again to stop it.
3. Click ● (green) to stop the Profiler.

Figure 9: Results of profiling section of code using breakpoints



```
All Functions
   @0x0c010024 < ENTRY POINT >                     27 (.034164%)   001
   struct LL_Node *GetNewNode()  @0x            26299 (33.2776%)   004
   struct LL_Node *MallocNode()  @0x            26299 (33.2776%)   008
   void BuildList()  @0x0c010050                   27 (.034164%)   002
   void CleanUp()  @0x0c010044                     26 (.032899%)   002
   void FreeHeap()  @0x0c010148                    26 (.032899%)   010
   void Initialise()  @0x0c010034                  27 (.034164%)   001
   void InsertNode(struct LL_Node *,            26298 (33.2763%)   007
   TOTALS                                       79029
```

T1:P1 File: C:/Current_projects\.../profexam.c

```
                                          int main(void)
                                          {
                    0x0C010020              while(1)
                                            {
27 (.034164%)       0x0C010024                 Initialise();
27 (.034164%)       0x0C010028                 BuildList();
```

T1:P1

```
                    0x0C010022      0009         nop
27 (.034164%)       0x0C010024      B006         bsr       Initialise
                    0x0C010026      0009   <s>   nop
27 (.034164%)       0x0C010028      B012         bsr       BuildList
                    0x0C01002A      0009   <s>   nop
26 (.032899%)       0x0C01002C      B00A         bsr       CleanUp
                    0x0C01002E      0009   <s>   nop
                    0x0C010030      AFF8         bra       $0c010024
                    0x0C010032      0009   <s>   nop
27 (.034164%)       Initialise      E300         mov       #$00,     r3
                    0x0C010036      D202         mov.l     #List,    r2
```

NOTE: *You can set the Profiler to execute only one pass between breakpoints from the Profiler shortcut menu. You must do this before enabling the Profiler.*

The Hits column and the hits scores in the Source and Disassembly regions show all the functions called from the section of code during the profile and the number of hits on all the functions within that section of code.

In the example program, the linked list of numbers is built up to 1000 and then cleared. This is shown by the functions marked with a hot spot that have approximately 1000 more hits than the other functions called during the profile. The hot spot functions are called 1000 times for each loop of the program while the other functions are called only once. The program was in its 27th loop when the Profiler was stopped.

Appendix A: Profiler’s shortcut menu

Table 2: The Profiler’s shortcut menu

Select:	To:
File	Load or save profile information. Profiles are saved using the extension .prf.
Enable Profiler	Start or stop the profiler.
One Pass Between Breakpoints	Set one pass between a Profiler start breakpoint and a profiler stop breakpoint.
Remove All Profiler Breakpoints	Remove all Profiler breakpoints.
Trace Tree Profile Display	Display a tree showing each function, the functions that called it, and the functions it called. Trace profiling only.
Function Profile Display	Display a simple list of the functions in your program in the Profiler window.
Function Profile Filter	Arrange the view to show one of the following: all functions, all tagged functions, or all untagged functions.
Untag All	Untag all currently tagged functions.
Sort	Arrange the column view of the active Function Profile Display.
Source Display	View your program’s original source code.
Disassembly Display	View your program at instruction level (assembly code).
Rename Function...	Enter a new name for a specific function.
Profiler Display Setup...	Specify the profile display options.

Select:	To:
Optimize...	Instruction cache optimization. Trace profiling only.
Setup...	Specify options for statistical profiling or function trace profiling.

Appendix B: Profiler File Format

This appendix describes the Profiler File Format. The following tables of data structures that define the file format are in the order that they appear in the file.

The file format consists of six structures:

1. Profiler File ID
   This is the label at the start of the file which initially identifies it as a profiler file.
2. File Header
   This is the file's header which contains global information about the profile.
3. Overlay Area Data
   This is a description of an overlay area.
4. Function Data
   This contains data of a function that was called during the profile.
5. Trace Function Call
   This describes a function call which has its own function data unique to that particular call path.
6. Trace Function Return
   This describes a function return.

The position of the *Trace Function Call/Return* structures in the file defines the function's call/return relationship.

If the file represents a Statistical Profile as opposed to a Trace Profile there will be no *Trace Function Call* or *Trace Function Return* structures in the file.

The *File Header* indicates the type of profile. If it is a Statistical Profile the *Number Of Function Calls* and *Trace Information Offset* will be set to zero along with all the Counter data. In this case only the Function Hits data is valid. If the file represents a Trace Profile all the data is valid.

Incorrect counter data (see *Function Data Structure and Trace Function Call Structure*) can occur when there is more than one Profile Event (JSR, RTS, Interrupt, Exception, RTE, BSR, BSRF) in the SH4's pipeline. This means there is no way of knowing which event incremented which counter and by what amount. This is an anomaly of the SH4's tracing hardware and the effect on the Profiler's results is usually negligible.

The *Counter Description Table* describes all the counters available in he SH4's ASE mode debug facility.

Profiler File ID Structure

Byte	Type	Profiler File ID
0	Byte	0x50 'P'
1	Byte	0x52 'R'
2	Byte	0x4f 'O'
3	Byte	0x46 'F'
4	Byte	0x49 'I'
5	Byte	0x4c 'L'
6	Byte	0x45 'E'
7	Byte	0x52 'R'
8	Byte	0x20 ' '
9	Byte	0x46 'F'
10	Byte	0x49 'I'
11	Byte	0x4c 'L'
12	Byte	0x45 'E'
13	Byte	0x3a ':'

File Header Structure

Byte	Type	File Header
0 1 2 3	Unsigned Long	(LSB) Size of Header to follow. (0x00000055) (MSB)
4	Byte	File Version
5 6	2 bytes	Spare Bytes
7	Byte	Instruction Cache : 0x01 Enabled (8k) , 0x00 Disabled.
8	Byte	Day (0 - 31)
9	Byte	Month (0 - 12)
10	Byte	Year eg. 98 - 99 - 00 - 01
11	Byte	Hour (0 - 24)
12	Byte	Min (0 - 60)
13	Byte	Second (0 - 60)
14	Byte	Type of Profile 0x01 - Statistical Profile - in this mode only the Function Hits are valid. All the counter data in every structure will be zero. 0x02 - Trace Profile - all counters and hits are valid.
15 16 17 18 19 20 21 22	Unsigned Quadword	(LSB) Overall accumulative total number of Function Hits (MSB)

Byte	Type	File Header
23 24 25 26 27 28 29 30	Unsigned Quadword	(LSB) Overall accumulative total of Counter 1 (MSB)
31 32 34 35 36 37 38	Unsigned Quadword	(LSB) Overall accumulative total of Counter 1 minus the functions it called. (MSB)
39 40 41 42 43 45 46	Unsigned Quadword	(LSB) Overall accumulative total of Counter 2 (MSB)
47 48 49 50 51 52 53 54	Unsigned Quadword	(LSB) Overall accumulative total of Counter 2 minus the functions it called. (MSB)
55	Byte	What Counter 1 represents, see the Counter Description Table at the end of the document.
56	Byte	What Counter 2 represents, see the Counter Description Table at the end of the document.
57 58 59 60	Unsigned Long	(LSB) Number of Overlays (MSB)

Byte	Type	File Header
61 62 63 64	Unsigned Long	(LSB) Number of functions (MSB)
65 66 67 68	Unsigned Long	(LSB) Number of function calls. If Byte 14 is set to 0x01 (for Statistical Profile) then this will be set to zero. (MSB)
69 70 71 72	Unsigned Long	(LSB) Overlay Information Offset Offset into this file from the End of the Header to the Overlay Information. This is only valid if the Number Of Overlays field is greater than zero. (MSB)
73 74 75 76	Unsigned Long	(LSB) Function Information Offset Offset into this file from the End of the Header to the Function Information. (MSB)
77 78 79 80	Unsigned Long	(LSB) Trace Information Offset Offset into this file from the End of the Header to the Trace Information (function calls/returns). If Byte 14 is set to 0x01 (for Statistical Profile) then this will be set to zero. (MSB)
81 82 83 84 85 86 87 88	8 bytes	Spare.

Overlay Area Data Structure

The Overlay Data that follows is a list of Overlays that exist within the program. The data is repeated *Number Of Overlays* times. *Number Of Overlays* can be found in the File Header.

Byte	Type	Overlay Area Data
0 1 2 3	Unsigned Long	(LSB) Size of Overlay Area Data to follow. (MSB)
4 5 6 7	Unsigned Long	(LSB) Overlay Area Index ID. (MSB)
8 9 10 11	Unsigned Long	(LSB) Start Address of Overlay Area. (MSB)
12 13 14 15	Unsigned Long	(LSB) End Address of Overlay Area. (MSB)
16 17 18 19	Unsigned Long	(LSB) Current Overlay ID (at the time of saving). (MSB)

Function Data Structure

The Function Data that follows is a list of functions called during the profile with Accumulative Totals for that function regardless of call path. The data is repeated *Number Of Functions* times. *Number Of Functions* can be found in the File Header.

Byte	Type	Function Data
0 1 2 3	Unsigned Long	(LSB) Size Of Function Data to Follow. (MSB)
4	Byte	Function Description. 0x00 - Unknown. 0x01 - Function. 0x02 - Interrupt/Exception (unknown VBR + ?). 0x03 - Exception VBR + 0x100. 0x04 - Exception VBR + 0x400. 0x05 - Interrupt VBR + 0x600. 0x06 - Exception DBR. 0x07 - User Defined Block.
5 6	Unsigned Word	Attributes (Bit Field) 0x00000001 - This function has possible incorrect counter data. 0x00000002 - This function was Tagged (Selected) by the user. All other bit-fields will be 0.
7 8	Unsigned Word	(LSB) Cache Line Number. 0 - 256 (256 lines * 32 Bytes = 8k) , 0xFFFF Non Cacheable (MSB)
9 10 11 12	Unsigned Long	(LSB) Minimum Program Counter of Function Scope. (MSB)
13 14 15 16	Unsigned Long	(LSB) Max Program Counter of Function Scope. (MSB)
17 18 19 20	Unsigned Long	(LSB) Overlay Area Index ID ID 0xFFFFFFFF = Not part of an overlay. (MSB)

Byte	Type	Function Data
21 22 23 24	Unsigned Long	(LSB) Overlay ID ID 0xFFFFFFFF = Not an Overlay. (MSB)
25 26 27 28	Unsigned Long	(LSB) Function Group ID ID 0xFFFFFFFF = No Group. (MSB)
29 30 31 32 33 34 35 36	8 Bytes	Spare.
37 38 39 40 41 42 43 44	Unsigned Quadword	(LSB) Accumulative total number of Function Hits for this function. (MSB)
45 46 47 48 49 50 51 52	Unsigned Quadword	(LSB) Accumulative total of Counter 1 for this function. (MSB)
53 54 55 56 57 58 59 60	Unsigned Quadword	(LSB) Accumulative total of Counter 1 minus the functions it called for this function. (MSB)

Byte	Type	Function Data
61 62 63 64 65 67 68	Unsigned Quadword	(LSB) Accumulative total of Counter 2 for this function. (MSB)
69 70 71 72 73 74 75 76	Unsigned Quadword	(LSB) Accumulative total of Counter 2 minus the functions it called for this function. (MSB)
77 78	Word	(LSB) The size of the following function name. ('n1') (MSB)
79	String	Function Name. The string is of the size specified in the previous field.
79 +'n1'	Word	(LSB) The size of the following Linker Label. ('n2') (MSB)
79 +'n1' + 2	String	Linker Label. The string is of size specified in the previous field. The Linker Label is the label that was used to link this function.
79 +'n1' +2+'n2'	Word	(LSB) The size of the following Qualified Name. (MSB)
79 +'n1' +2 +'n2' +2	String	Qualified Name. The string is of size specified in the previous field. The Qualified Name is used to specify a function in greater detail (filename, class etc.) in the event of a duplicate function name.

Trace Function Call Structure

The Function Trace Data (function calls/returns) consists of two data structures *Trace Function Call* and *Trace Function Return*. To distinguish between them the first byte after the data size is 0xFF for a function call and 0x00 for a return. There is a file offset to this data that is stored in the File Header. Also stored in the File Header is the *Number of Function Calls*. The number of following Trace Function Calls is coupled with an equal amount of Trace Function Returns.

Byte	Type	Trace Function Call
0 1 2 3	Unsigned Long	(LSB) Size of Trace Function Call Data to follow. (MSB)
4	Byte	0xFF Function Call identifier.
5 6 7 8	Unsigned Long	(LSB) Unique Call Number. This number is present in the matching 'Return From Function' data. (MSB)
9	Byte	Function Description. 0x00 - Unknown. 0x01 - Function. 0x02 - Interrupt/Exception (unknown VBR +). 0x03 - Exception VBR + 0x100. 0x04 - Exception VBR + 0x400. 0x05 - Interrupt VBR + 0x600. 0x06 - Exception DBR. 0x07 - User Defined Block.
10 11	Unsigned Long	Attributes (Bit Field) 0x00000001 - This function has possible incorrect counter data. All other bit-fields will be 0.
12 13	Unsigned Word	(LSB) Cache Line Number. 0 - 256 (256 lines * 32 Bytes = 8k), 0xFFFF(-1) Non Cacheable (MSB)
14 15 16 17	Unsigned Long	(LSB) Minimum Program Counter of Function Scope. (MSB)

Byte	Type	Trace Function Call
18 19 20 21	Unsigned Long	(LSB) Max Program Counter of Function Scope. (MSB)
22 23 24 25	Unsigned Long	(LSB) Overlay Area Index ID ID 0xFFFFFFFF = Not part of any overlay. (MSB)
26 27 28 29	Unsigned Long	(LSB) Overlay ID ID 0xFFFFFFFF = Not an overlay. (MSB)
30 31 32 33	Unsigned Long	(LSB) Function Group ID ID 0xFFFFFFFF = No Group. (MSB)
34 35 36 37 38 39 40 41	8 Bytes	Spare.
42 43 44 45 46 47 48 49	Unsigned Quadword	(LSB) Accumulative total number of Function Hits for this function with this call path. (MSB)
50 51 52 53 54 55 56 57	Unsigned Quadword)	(LSB) Accumulative total of Counter 1 for this function with this call path. (MSB)

Byte	Type	Trace Function Call
58 59 60 61 62 63 64 65	Unsigned Quadword	(LSB) Accumulative total of Counter 1 minus the functions it called for this function with this call path. (MSB)
66 67 68 69 70 71 72 73	Unsigned Quadword	(LSB) Accumulative Total of Counter 2 for this function with this call path. (MSB)
74 75 76 77 78 79 80 81	Unsigned Quadword	(LSB) Accumulative Total of Counter 2 minus the functions it called for this function with this call path. (MSB)
82 83	Word	(LSB) The Size of the Following Function Name. ('n1') (MSB)
84	String	Function Name. The string is of size specified in the previous field.
84 +'n1'	Word	(LSB) The Size of the following Linker Label. ('n2') (MSB)
84 +'n1' + 2	String	Linker Label. The String is of size specified in the previous field. The Linker Label is the label that was used to link this function.
84 +'n1' +2+'n2'	Word	(LSB) The Size of the following Qualified Name. (MSB)

Byte	Type	Trace Function Call
84 +'n1' +2 +'n2' +2	String	Qualified Name. The string is of size specified in the previous field. The Qualified Name is used to specify a function in greater detail (filename, class etc.) in the event of a duplicate function name.

Trace Function Return Structure

Byte	Type	Trace Function Return
0 1 2 3	Unsigned Long	(LSB) Size Of Trace Function Return Data to Follow. (0x00000005) (MSB)
4	Byte	0x00 Function Return Identifier.
5 6 7 8	Unsigned Long	(LSB) Unique Call Number. This number is to ensure that the return Unique Number matches the Call Unique Number. (MSB)

Counter Description Table

Counter Description Number	Counter Description	Count/Cycles
0x01	Operand access (read/with cache)	Count
0x02	Operand access (write/with cache)	Count
0x03	UTLB miss	Count
0x04	Operand cache read miss	Count
0x05	Operand cache write miss	Count
0x06	Instruction fetch (with cache) - 2 instructions fetched simultaneously	Count
0x07	Instruction TLB miss.	Count
0x08	Instruction cache miss	Count
0x09	All operand access	Count
0x0a	All instruction access - 2 instructions fetched simultaneously	Count
0x0b	On-chip RAM operand access	Count
0x0c	On-chip RAM instruction access	Count
0x0d	On-chip I/O space access	Count
0x0e	Operand access (read + write/with cache)	Count
0x0f	Operand cache read + write miss	Count
0x10	Branch instruction	Count
0x11	Branch taken	Count
0x12	BSR/BSRF/JSR	Count
0x13	Instruction execution	Count
0x14	2- instruction simultaneous execution	Count
0x15	FPU instruction execution	Count
0x16	Interrupt (normal)	Count

Counter Description Number	Counter Description	Count/Cycles
0x17	Interrupt (NMI)	Count
0x18	TRAPA instruction execution	Count
0x19	UBC-A match	Count
0x1a	UBC-B match	Cycles
0x21	Instruction cache fill	Cycles
0x22	Operand cache fill	Cycles
0x23	Elapsed time	Cycles
0x24	Pipeline freeze (by cache miss/instruction)	Cycles
0x25	Pipeline freeze (by cache miss/data)	Cycles
0x27	Pipeline freeze (by branch instruction)	Cycles
0x28	Pipeline freeze (by CPU register)	Cycles
0x29	Pipeline freeze (by FPU)	Cycles

Cross Products Ltd
23 The Calls, Leeds, West Yorkshire, LS2 7EH
telephone: +44 113 242 9814
facsimile: +44 113 242 6163
www.crossproducts.co.uk
email sales: enquiry@crossproducts.co.uk
email support: support@crossproducts.co.uk