Freescale Semiconductor
Application Note

Document Number: AN4250
Rev. 0, 01/2011

Software Analysis Tips and Tricks Using CodeWarrior for StarCore DSPs

by *DevTech Customer Engineering*
Freescale Semiconductor, Inc.
Austin, TX

This document covers some of the new features in the software analysis tools that are part of the Freescale Eclipse-based CodeWarrior plug-in. These tools help track and measure a DSP application's code flow and help the user debug the application faster. They also help determine where the performance bottlenecks are in the code.

This application note is based on real customer experience in the field and provides more specific information, above that provided by the guidelines presented in the manuals.

This document assumes the reader is familiar with the Eclipse IDE. It also assumes that the user understands the CodeWarrior Software Analysis GUI and the concepts behind it.

All of the results and screen captures are based on the CodeWarrior for StarCore DSPs 10.1.5 release. For later releases, there might be minor GUI changes, but the operation of the tools should be similar.

Contents

1 Use Counter Points

Counter points, as their name implies, count events between two specific points in the code as it executes. The results are computed by searching the collected trace data, using only the information between the counter point addresses.

1.1 Activating the Feature

This feature is available from the **Trace and Profile Results** view, as shown in Figure 1.

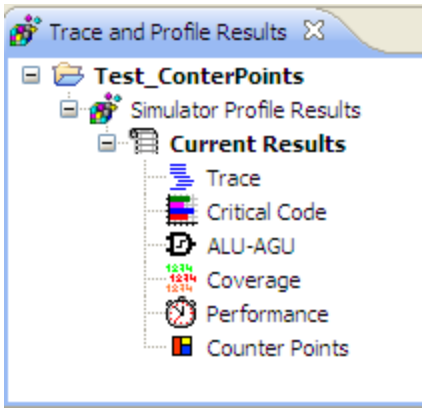


Figure 1. Location of Counter Points Control.

Once the trace data has been collected and the feature is activated, the results can be displayed by double-clicking on the `Counter Points` item in this view.

1.2 When to Use It

Counter points is a valuable feature when the user wants to get the trace data for functions that are executed cyclically in the DSP applications.

Taking into account the application setup and algorithm, the same function, when called multiple times, might have different code and data coverage. This feature allows collecting and filtering all the relevant information for the selected function or functions.

NOTE

One way to gather coverage information would be to check and filter the trace dump data manually. However, this is a time-consuming method and not suitable for fast debugging purposes nor performance/regression validation tests.

1.3 How to Set Up the Counter Points

As an example of how to use counter points, assume the application has the following code:

```
 int main()

{

      int i;

      for (i=0;i<10;i++)

            func2();

      return func1();

}
```

Assume that the developer wants to trace data for the function `func2()` each time that it is called.

In order to setup the counter point feature, right-click on the marker bar to the left side of the C/C++ editor view and select **Toggle counter point** from the drop-down menu that appears.

If this operation is successful, a blue arrow appears in the marker bar. It points to the source line which acts the starting point for the trace statistics collection, as shown in Figure 2.

To designate the end of where statistics should be collected, right-click in the marker bar at the desired source statement and choose **Toggle counter point** again.

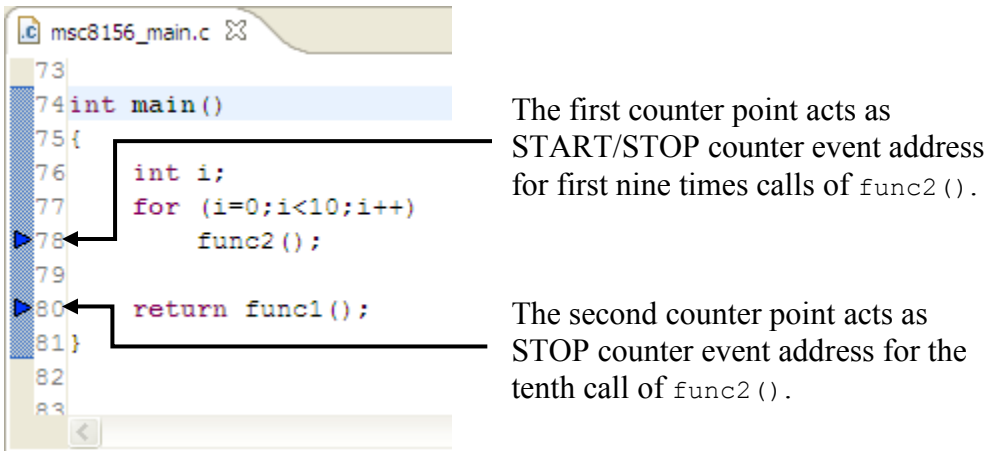


Figure 2. Adding Counter Points.

NOTE

The counter points results are computed based on the trace data dump. The counter points MUST be placed only on code that generates trace information. Do not set counter points on the lines containing comments, brackets, or variable declarations.

NOTE

When analyzing a function generated from optimized code, verify in the assembly language listing that the first instruction into a VLES block is the jump/call to the specific function. In other words, the change of flow (COF) must be the first instruction in a VLES. Otherwise, the results might not be correct.

If the application requires that more portions of the code must be analyzed, then multiple start and stop counter points can be set. The analysis is carried out on each of the counter point pairs.

NOTE

There is no limit on the number of counter points that can be used in a project. The analysis is done offline, after the trace data has been downloaded into the host PC. Nevertheless, a large number of counter points will adversely affect the speed at which CodeWarrior displays the collected data.

At any time, the counter points can be disabled either by right-clicking in the marker bar, or by right-clicking the counter point in the **Analysispoints** view and choosing **Disable** (Figure 3).

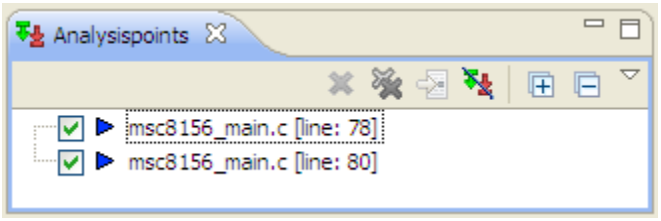


Figure 3. Right-Click on the Counter Point in the Analysispoints View to Disable/Delete It.

The change is visible immediately in the **Editor** view.

NOTE

The **Analysispoints** view is displayed by selecting **Window > Show View > Other > Analysis > Analysispoints**.

1.4 Obtaining Results

To collect the trace data with the hardware/simulator, the Trace and Profile collection mechanism must be enabled and a debug session executed first. Once these requirements have been fulfilled, select the `Counter Points` dataset from the **Trace and Profile** view as shown in Figure 1.

It takes several seconds to parse the trace data dump for the counter point addresses. Once the delta statistics computation completes, the results are displayed in the **CpResultsEditor** view (Figure 4).

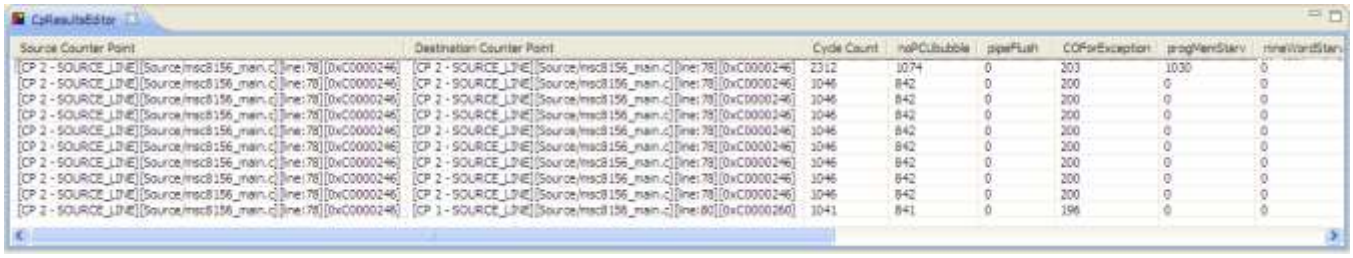


Figure 4. The Counter Point Results Display.

NOTE

For hardware debugging, fewer columns are displayed since the trace on hardware is limited to a specific number of counted events. The available columns depend whether the On-Chip Emulator (OCE) or the Debugging and Profiling Unit (DPU) is used to collect the trace. In case of the DPU choice, it also depends of the actual DPU settings.

The **Source Counter Point** column in this view indicates the start event of the measurement. In this particular case, the starting point of the count is the jump to 0xC0000246 (the `func2()` function) which is located in the `msc8156_main.c` file. The counter point is installed at line number 78.

The **Destination Counter Point** displays the stop event of the measurement. For the first line shown in Figure 4, the stop event is triggered by a new call to `func2()`.

For the last entry of the table displayed in Figure 4, the stop event is triggered by the call to `func1()` located at the program address 0xC0000260. This counter point is installed at line 80 in `msc8156_main.c` source file.

You can also count the event by enabling only one counter point as shown in Figure 5:

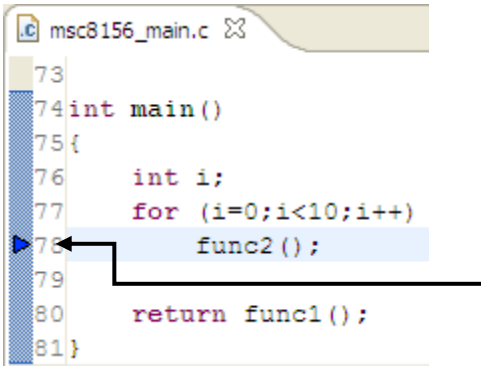


Figure 5. Setting One Counter Point.

NOTE

In this situation, the tenth call is not counted because there is no event to mark the stop of the measurement. Still, the counter points can be used in this scenario but be aware that displayed number of calls is short one count.

NOTE

To avoid erroneous results in the analysis, it is best to use a pair of counter points (one marking a start address and the other one marking a stop address) for every code section under analysis.

2 Facts About Traces

For complex applications where the real-time behavior of the system is critical for the algorithm's accuracy, the use of the Trace and Profile feature could cause an application failure, or seriously degrade the responsiveness of a debug session. This section discusses different techniques that can reduce the intrusiveness of the trace for these situations.

2.1 How Trace Collection Works

To understand how these techniques work, a description of how the trace and profiling feature operates is helpful. At hardware level, the unit responsible for data/code events collection is the DPU.

NOTE

Details about the DPU operation can be found in the *MSC8156 SC3850 DSP Subsystem Reference Manual*. This document only covers the key aspects the DPU operation that facilitates the user understanding of how it generates trace data. For any other specific information, check the manual.

The DPU writes the trace information that is generated by either itself or the On Chip Emulator (OCE), to a Virtual Trace Buffer (VTB) as Figure 6 shows. The VTB can reside in system memory or in internal or external memory, as specified by the user.

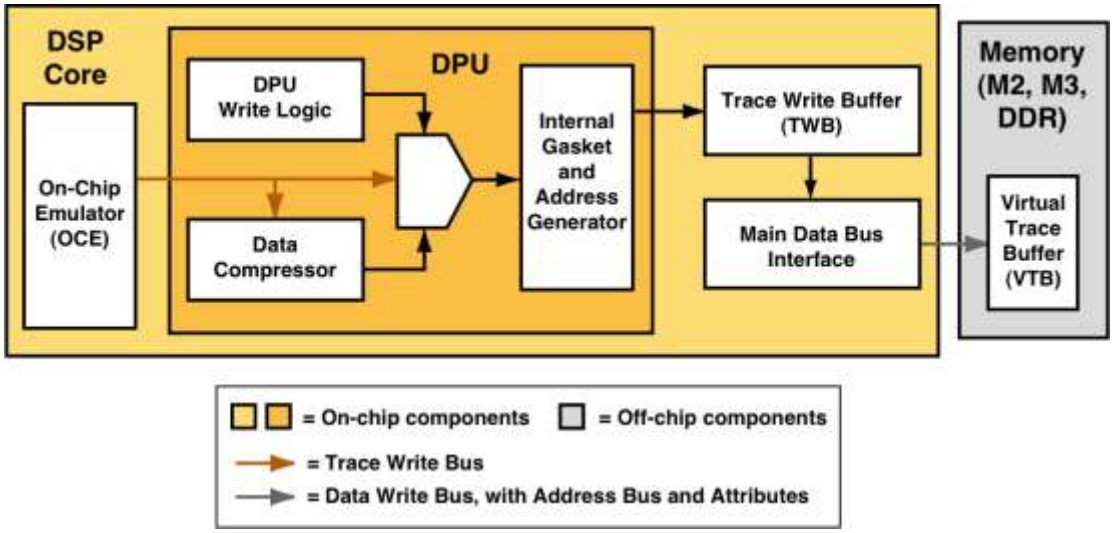


Figure 6. How Profile Data is Written to the VTB.

The structure of the DPU counters is shown in Figure 7. The DPU consists of six down-count counters, grouped in two triads. A *triad*, as its name implies, consists of a group of three 31-bit resolution counters.

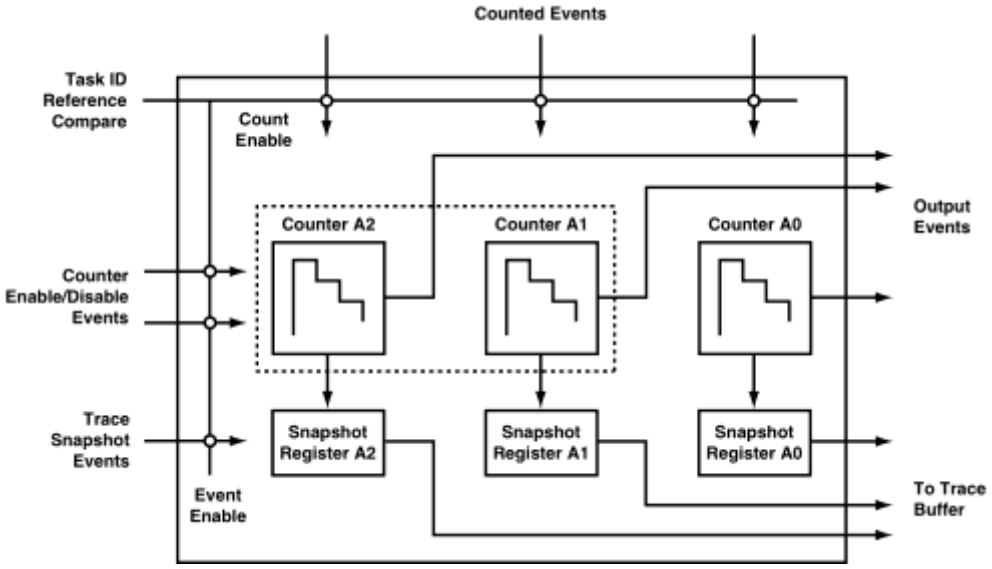


Figure 7. Structure of the Counters in DPU A Triad.

Each counter triad can operate in three distinct modes:

- **One-shot counting mode**—The counters count down from a predefined value (or from their reset value, 0x7FFFFFFF) to zero, stop counting, and generate an event. For example, they can cause the core to enter debug mode, generate a debug exception, or generate an interrupt debug A or an interrupt debug B to the EPIC. The counter is disabled when it stops counting.
- **Trace mode**—When a trace event arrives from the trace logic, the value of each counter is copied to its respective snapshot register, while the counters continue counting. The snapshot registers are then written to the VTB. If a counter reaches zero, it continues to count, starting from the reset value (0x7FFFFFFF). That is, when the counter value becomes zero, it "rolls over" to the reset value and resumes counting down.
- **Extension mode**—When configured individually, the A2 and B2 counters operate in extension mode. For these two counters, the counter value overflows when it reaches zero and continues to count. The A1 and B1 counters can be programmed to count the overflows of counters A2 and B2, respectively. In this case, counters A1 and B1 must operate in one-shot counting mode, so that the counter-pair in each triad implement a 62-bit virtual counter. This mode is used primarily to count clock cycles for extended periods. This mode is valid only for the StarCore MSC8156 part. (See the *MSC8156 SC3850 DSP Subsystem Reference* Manual for details).

NOTE

To manually customize the counters, select either the **DPU Triad A** or **DPU Triad B Settings** tab.

NOTE

Once the trace counters are programmed to function in a specific mode, their configuration cannot be changed during program operation.

The DPU write accesses are buffered in the Trace Write Buffer (TWB), and then written through the main data bus interface in bursts. The lowest priority bus requests are used when writing into the first unwritten address of the VTB. When TWB becomes full, it raises the priority of its bus request. At the first cycle after this priority change, the DPU generates a core stall request.

When the writes into the VTB reaches the **VTB Trace Event Request Address**, it triggers a debug interrupt and the data are downloaded into the host PC.

This mechanism presents some bottlenecks that are analyzed in the sections that follow.

2.2 Why the Trace Download Rate Can Be Very Low

To perform its performance analysis, the CodeWarrior debugger needs the trace data collected by DPU. Once the trace data has been saved into the VTB buffer, it is downloaded into the host PC. The data transfer supports three interfaces, termed methods: JTAG, HSST and HEAT.

The interface used can be chosen in the **Trace and Profile** tab, within the **Trace offload method** group (Figure 8).

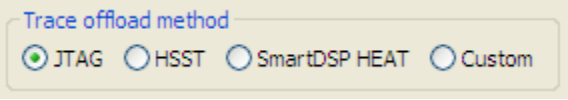


Figure 8. The Choice of Trace Download Methods.

NOTE

The **Custom** method choice currently uses the JTAG interface to perform a download. It is reserved for future use.

The default method/interface for the trace download is JTAG. This is the easiest method since it does not involve any other settings or special application builds. In case of the HSST and HEAT methods, there are some constraints, such as:

1. The user application must be linked with Freescale-specific libraries.
2. Specific C/C++ source files must be used to make the appropriate hardware target settings.
3. The HEAT method requires the use of the Freescale SmartDSP operating system (SDOS).
4. For HEAT, an external application must be executed to obtain the trace.

For those cases where the application cannot meet any of the requirements described in items one through four, the only suitable method is the JTAG choice. However, the JTAG has a limited download speed, due to the JTAG connection's limited bandwidth.

If the application design permits it, Freescale strongly advises the use of the other two methods: HSST and HEAT.

Figure 9 and Figure 10 compare the data transfer rates between the JTAG, HSST and HEAT methods, taking into account the VTB buffer size. The small buffer holds 256 bytes, the medium-sized buffer contains 28.5 KB, the large stores 390 KB, and the big stores 10 MB.

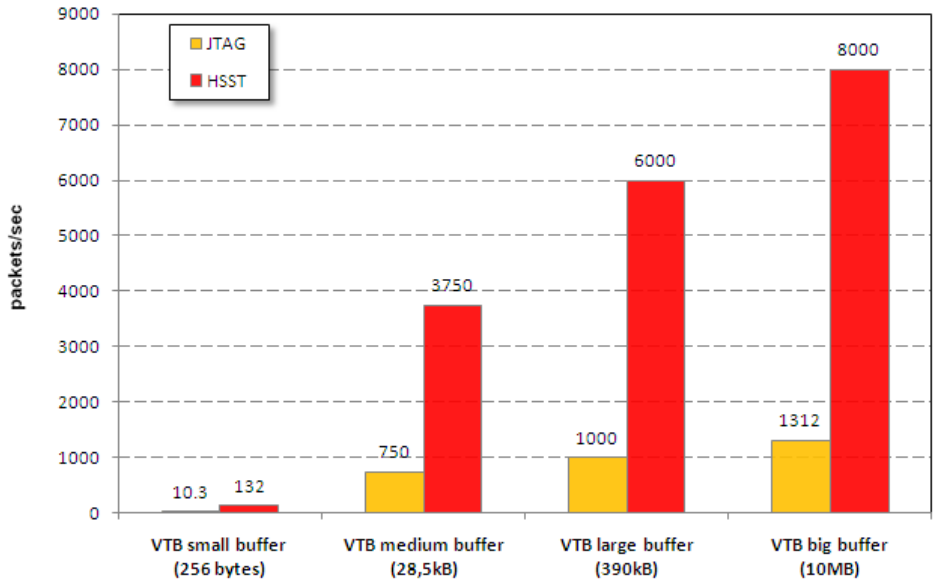


Figure 9. JTAG Versus HSST Download Rates.

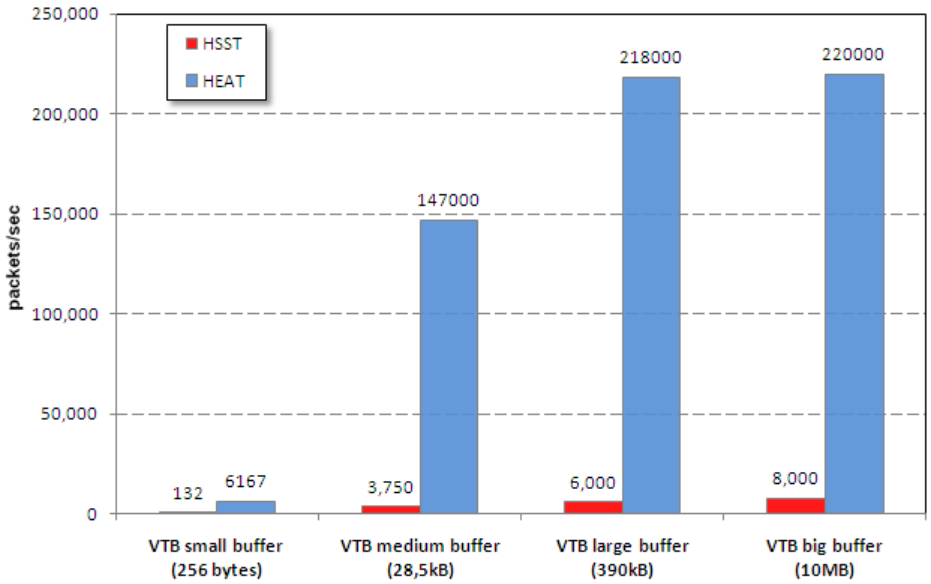


Figure 10. HSST Versus HEAT Download Rates.

As can be seen from the figures, the HEAT interface is about 160 times faster than JTAG, and it is 25 times faster than HSST. The following notes should help decide which method to use:

- For the choice of HEAT, the optimal buffer size is considered to be 10 MB. Buffer sizes larger than that size are affected adversely due to the host PC's hard disk data management, the level of disk fragmentation, or the overhead of other host PC tasks.
- The memory type (M2/M3/DDR) where the VTB is stored has no influence upon the download speed. Experiments were carried out for the same application using all three types of memory, and the results for the same download method are identical.
- For best performance—that is, for the shortest download times—the recommended method is HEAT. If the HEAT choice is not possible because the SDOS/external application requirements, HSST is the recommended download method.
- The JTAG download method should be used only in case of small- and medium-sized (less than 30 KB) VTB buffers, and where the effort of configuring the setup for HEAT/HSST downloads is larger than the time lost for downloads.

2.3 Why the Trace Can Introduce Large Latencies in the System

When Trace and Profile has been enabled, the DSP application can take much more time to complete. Even worse, the real-time behavior of the application is affected such that the algorithm does not execute as designed (especially in the case of multicore applications), and the application yields wrong results.

There are many aspects that negatively impact an application's performance, and can lead towards a catastrophic application failure. What follows is a discussion of some these issues and what might be done to reduce their effects.

2.3.1 Degradation Due to Memory Bus Accesses

This problem is caused by the writes from the TWB to VTB (see Figure 6). The TWB is a small buffer through which 32-byte DPU write accesses are buffered before being written into the VTB. These writes are transferred via the main data bus interface at the lowest priority.

Under normal conditions, the TWB sends a request to the Data Control Unit (DCU) which treats it with lowest priority. When there are no other requests for the bus pending, the DCU grants the TWB access to bus so that it writes its data into the VTB.

If the TWB becomes full, then the request to access the data bus changes from low to high priority. A cycle later, the DPU generates a core hold request. This stops the normal flow of the application in order to allow the DPU to write the data into the memory. These stalls can severely impact the application performance.

In order to limit the number of stalls that occur, count only the essential events, or one event at a time. This can be done by configuring the DPU triads in **Trace and Profile**'s advanced settings group to **Use Individual Counter settings** (Figure 11).

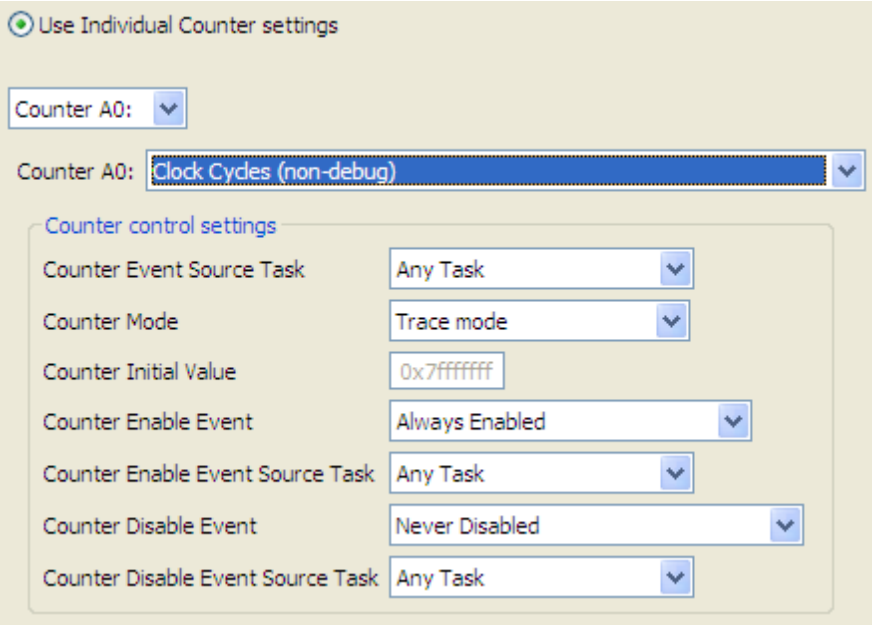


Figure 11. Adjusting the Individual Counter Settings.

NOTE

The penalties due the data transfer between TWB and VTB can be analyzed by configuring the DPU to trace the `Events on the External Buses`. Counters A1/B1 can trace the events generated by `Data transfer to the VTB (TWB write accesses)`. The counter values can be used to evaluate the impact of this issue on the application.

2.3.2 Degradation Due to VTB Write Cycles

When data is written to VTB buffer, the first write (address access) take seven cycles to set up, while the next operation, a write of a Virtual Bit Resolution (VBR), takes one cycle. The term VBR specifies how many bits comprise the burst transfer to the VTB, and it consists of 16 data bytes. The address access is a fixed operation and cannot be modified. However, the cost of this initial access can be reduced by having the DCU perform more data writes per transfer operation. That is, more VBRs should be written to the VTB during a transfer.

The amount of VBRs that can be written in a transfer is specified from the VTB **Advanced Settings** group (Figure 12). The choices are `1 VBR (16bytes)`, or `4 VBRs (64 bytes)`.

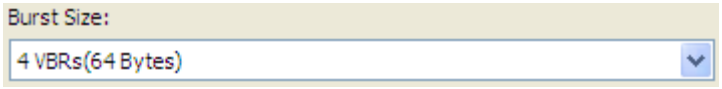


Figure 12. Specifying the Data Transfer Amount.

When the `1 VBR` option is chosen, each VTB transfer takes seven cycles for the address and one cycle for each VBR. For the `4 VBR` choice, each transfer to the VTB takes seven cycles for the address and one cycle for a 64 byte data transfer. Doing the math, a 64-byte transfer consumes 32 cycles when using the `1 VBR` option, and only 11 cycles when using the `4 VBR` option. Put another way, the `4 VBR` selection can triple the data transfer rate.

2.3.3 MSC8156 Memory Latency

Locating the VTB in slower memory affects the time needed to write the TWB data into it. Depending on the download method used (see section 2.2) a compromise must be made. Note that accesses to M2 memory require about 10 to 12 core cycles, M3 memory accesses require 54 core cycles, and DDR memory accesses require 91 to 95 core cycles. Therefore, use DDR memory to store the VTB only when you are using large data buffers.

2.4 Tips to Reduce the Impact of Traces

To minimize the effects of tracing on application performance, consider the following options:

- Count only the events of interest. Multiple debug sessions can be carried out automatically to obtain trace data for other events. Tracing the events individually means that TWB fills slowly and less writes to VTB are required.
- Use a VTB of the appropriate size for the download method used.
- Recall that the core stalls while the trace is downloaded. A larger-sized VTB generates less debug interrupts when transferring its content to the host PC. Therefore, the goal is to reduce the number of times the VTB must be emptied onto the host PC. Ideally, the VTB should have enough capacity to save all the events from a debug session, so that a VTB download occurs only when the debug session terminates. Practically, the goal should be to minimize the number of downloads required to transfer the trace data.
- Use M2/M3 memory when available.
- Use the Start/Stop trace points to take measurements only of the critical code. This results in less data that must be saved in the VTB.

3 Filtering Trace Data

This section discusses possible configuration options that have the software analysis module display only the data of interest. The CodeWarrior tools offer the means to automatically filter and display the trace data in accordance with the user requirements. Note that certain of these configuration options might not be visible in the GUI. For example, an `internal_func.txt` file can be used to establish a semi-automatic task that filters the trace data. Practically, the software analysis module uses information in this file when it excludes any specified functions/symbols. Knowing this, the file can be edited to modify the filtering behavior.

The `internal_func.txt` file can be found inside the CodeWarrior installation `{CW install dir}\analysis\1.1.0\lib\host\engine\modules\tagStream\internal_func.txt`.

To use this feature, from the **Trace and Profile** tab, select the option **Exclude internal libraries from performance statistics** (Figure 13).

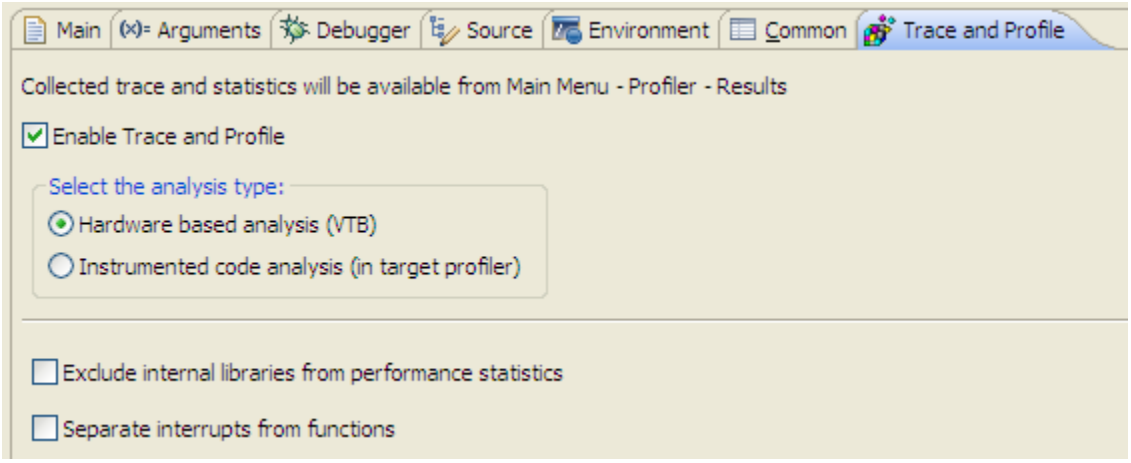


Figure 13. How to Activate the Software Trace Filter.

To test the technique, first build the same example program described in section 1.3 with the Codewarrior tools. Run the application and collect the trace. Display the trace data, which presents the application call tree shown in Figure 14:

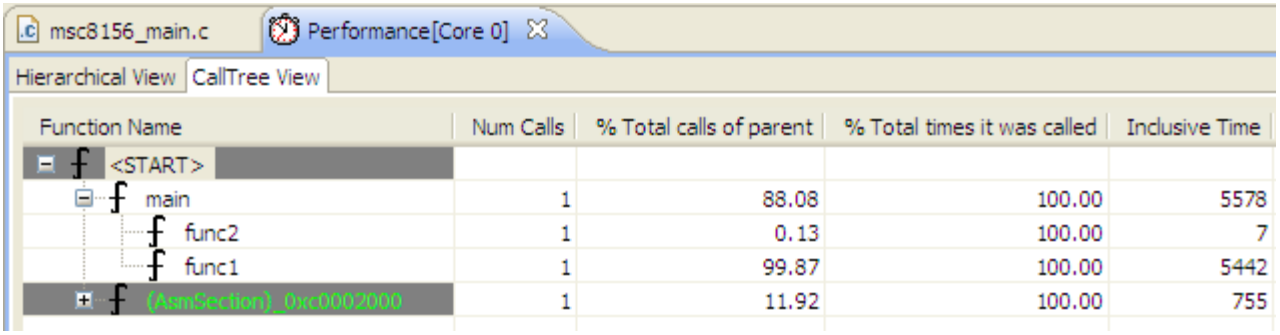


Figure 14. Display of Call Tree Without Filter Applied.

As can be seen in Figure 14, the `main()` function calls two other functions: `func1()` and `func2()`. Suppose that calls to `func2()` must be eliminated from the statistics.

This can be done in a series of steps as follows:

1. Identify the output file (`*.eln`) that implements `func2()`. In this example, which uses the built-in project template, the function implementation can be found in the file `msc8156_main.eln`.
2. Use the following script to extract the function-specific details that the filtering process requires. This is a Linux shell script which calls the `sc100-elfdump.exe` utility to extract the necessary information. For more details about the file format, consult the *CodeWarrior Development Studio for StarCore DSP Architectures Targeting Manual*, specifically the chapter, *StarCore DSP Utilities*.

```
func_file_temp="function_list_temp"

# dump the library files (.eln)

grep_list=""

for f in $( ls | grep .eln); do

    dump_file_name="${f%.*}"

    echo Processing $f

    ./sc100-elfdump.exe -a -m $f > "${f%.*}"

    grep_list="${dump_file_name} $grep_list"

done

echo
```

```
# get the FUNC records from the dumped files
echo Getting the FUNC records
grep FUNC $grep_list > $func_file_temp
```

3. Use the following script to extract the function-specific details required for the filtering process. The result of running the script should be a file that contains:

```
msc8156_main: 0x00000000 0x00000018 GLOBAL FUNC NONE  9 _func2{}
msc8156_main: 0x00000000 0x0000023a GLOBAL FUNC NONE 11 _func1{}
msc8156_main: 0x00000240 0x00000016 GLOBAL FUNC NONE 11  _main{}
```

4. Copy and paste the line for `_func2` into the end of the `internal_func.txt` file located in the directory:

```
{CW install
dir}\analysis\1.1.0\lib\host\engine\modules\tagStream\
```

NOTE

After pasting the line, edit it to resemble the lines above it. In this example, the line should resemble:

```
msc8156_main: 0x00000000 0x00000018 GLOBAL FUNC
9 _func2
```

5. Make sure the option **Exclude internal libraries from performance statistics** is checked and run the application again to collect the new trace data. Figure 15 shows the call tree with the filter in effect.

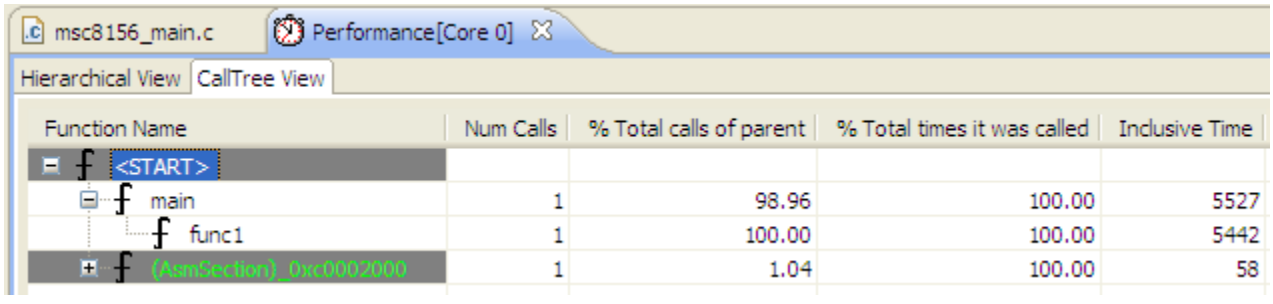


Figure 15. The Call Tree With the func2() Filter Applied.

Although this was a simple example, it demonstrates how functions can be filtered out of the trace by editing the `internal_func.txt` file.

4 Offline Trace

Trace and Profile can also be used to debug complex applications where random errors or crashes affect its operation. Random cases such as these are hard to be catch during a debug session and sometimes lots of consecutive runs are needed just to trigger such an error or crash. In such situations, the trace and profile feature offers users the ability to identify and isolate the context responsible for an error/crash appearance.

For example, a very common use case is to enable the trace and let it collect data until the error/crash occurs. Once the error/crash happens, the user application should stop the DSP and save the trace buffer data for later analysis.

Also, this procedure is very useful for custom products where there is no possibility to connect the DSP to a host PC (without the possibility to download the trace in real-time mode).

4.1 Configuring the Trace for Offline Use

The first step is to select the appropriate method for counting the events that might lead you to the root cause. An appropriate choice would be to select `breakdown of application cycles` in the **DPU Configuration Options**, since it monitors the overall application execution (Figure 16). Depending on the type of event suspected to cause the error/crash, other DPU or OCE settings might be chosen.

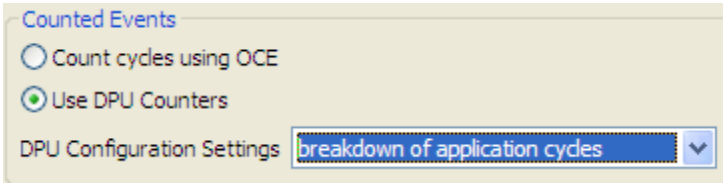


Figure 16. Option that Specifies How Events Are Counted.

The second step is to reserve a range of memory for the VTB (Figure 17). Usually this can be reserved in DDR, but other memory sections can be used. In this example, there is no need for a large buffer since the VTB acts as a circular buffer that is overwritten until the error/crash occurs.

Figure 17. How to Specify the Range of Addresses that the VTB Occupies.

The third critical step is to configure the VTB write mode to overwrite (Figure 18). Since the time interval when the error/crash occurs is undefined, the trace data needs to be collected while avoiding the possibility of filling the VTB.

Figure 18. Setting the VTB Contents to Be Overwritten.

After these three steps are completed, run the application. Note that the application code should be implemented so that when the error/crash happens, the data inside the VTB is not modified further.

4.2 Examining the Trace Data

When the error occurs, the DSP must be halted and the data from the VTB should be saved from the DSP into a file. For example, with the CodeWarrior for StarCore DSPs v10 debugger connected to the board, a target task can be used to transfer the trace data from the buffer into a file. For more information on how to set up the target task, consult the section, *Creating Target Task to Export Memory*, in the *CodeWarrior Development Studio for StarCore DSP Architectures Targeting Manual*.

If a CodeWarrior debugger connection is not available, an alternate way to save the data should be used. (For example, the Ethernet connection can be used to dump the data to a PC host.) After the file has been saved onto the host, the data can be examined to identify the context that caused the failure.

NOTE

For the CodeWarrior debugger to recognize the VTB data dump, the file's extension must be named `.rawtrace`. The procedure for importing a `*.rawtrace` file is described step-by-step in the *Profiling and Analysis Tools User Guide*, in the section *Importing Offline Trace Data*.

To import the raw data, proceed as follows:

1. Choose **File > Import > CodeWarrior > Software Analysis > Offline Trace**. Select the file with the raw trace data in it.

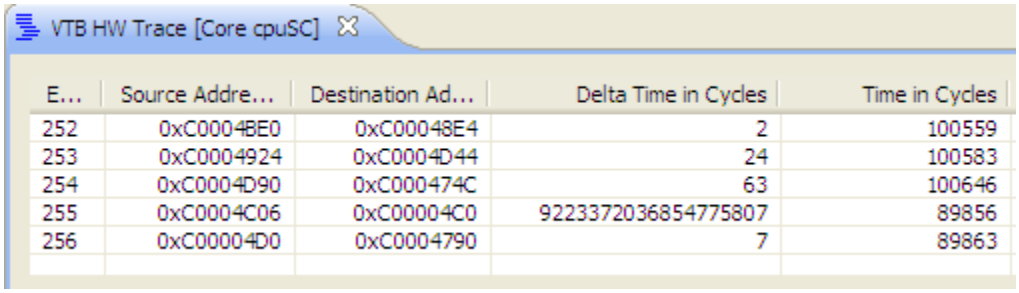
A new project titled **[Offline Trace]** appears in the **CodeWarrior Projects** view.

2. Open the **Trace_Profile_Results** folder in this project and open any trace file in it.

A **Trace and Profile** view appears.

3. In this view, open the **[Offline Trace]** project and the nested elements until the **Current Results** item appears.

4. Open the **Trace** item to display the trace information (Figure 19).



E...	Source Addre...	Destination Ad...	Delta Time in Cycles	Time in Cycles
252	0xC0004BE0	0xC00048E4	2	100559
253	0xC0004924	0xC0004D44	24	100583
254	0xC0004D90	0xC000474C	63	100646
255	0xC0004C06	0xC00004C0	9223372036854775807	89856
256	0xC00004D0	0xC0004790	7	89863

Figure 19. Identifying the Last Change in Flow from the Trace Data.

To identify the last change of flow in the code execution, study the **Delta Time in Cycles** column. The software analysis engine calculates the values based on the DPU counter values stored in the VTB.

When a huge number appears in the **Delta Time in Cycles** column, this is a sign that—between the two consecutive entries—something dramatic happened. The only logical explanation is to assume that the DPU counter values between these two events were captured at a different time intervals, and during that interval the VTB was overwritten. This huge value therefore points out where the application's last change in flow occurred.

NOTE

Unfortunately, the trace data is not designed to display any information regarding the context of any global/local variables or register values at the time of the crash. The user must determine the exact context that creates the error/crash. However, by pinpointing the area where the crash occurred, the trace display goes a long way in helping to solve the problem.

5 Instrumented Code Analysis

Beside VTB profiling, the CodeWarrior tool provide another means to analyze software, which is based on an instrumented code. Instrumented code is where software hooks in the application code enable some statistical information to be captured at runtime as each function is invoked. The **Trace and Profile** tab is used to enable this feature, as shown in Figure 20.

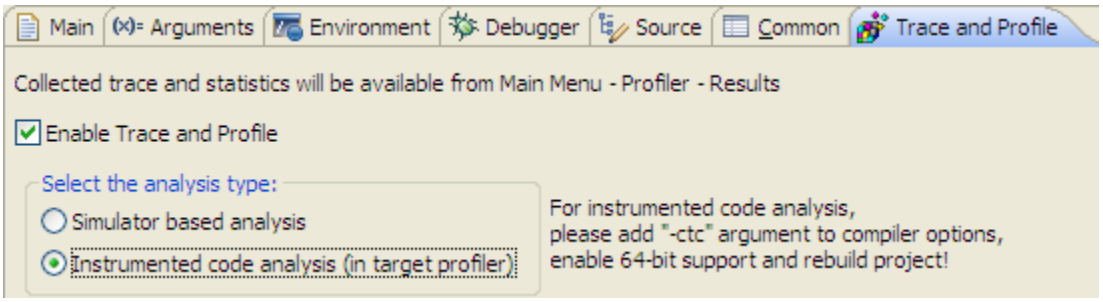


Figure 20. Enabling the Instrumented Code Feature.

NOTE

To configure the application so that it supplies tracking information to the profiler software, follow the steps described in the *Profiling and Analysis Tools User Guide*, inside the chapter *Using Code Instrumentation*.

Instrumented code analysis offers statistics only for the code that executes. This kind of information is generated by specifying the compiler argument `-ctc`. The information can also be used to validate the application flow. When this option is active, the compiler adds additional instructions into the executable file (`*.eld`) that help collect the required information.

One thing to watch out for is the different behavior for different targets when using instrumented code. For example, if one initially keeps/runs instrumented code (with the `-ctc` compiler flag enabled) and then switches the target to hardware-based analysis that uses the VTB, an error message appears. The error message states that VTB profiling is disabled. However, if the target is the simulator, there is only a warning and the collected trace data in the VTB contains the instrumentation information.

NOTE

The instrument code adds overhead that can alter the application/program's performance dramatically, and potentially can produce inaccurate results known as "heisenbugs". (Heisenbug is a play on the name of Werner Heisenberg, who proved that the act of measuring an experiment can affect its outcome.) At a minimum, instrumentation has a negative impact on the program execution, often by slowing it down.

The results are collected into a file named `ctc.txt`, which by default is located in the folder `{Project location}\Trace_Profile_Results`.

The beginning of the `ctc.txt` file contains overall code coverage statistics of the application:

```
Overall Summary (SC): 1 Source File     1 Include File
  60 blocks
        9  15.0%  covered
       51  85.0%  not covered
  225 statements
       38  16.9%  covered
      187  83.1%  not covered
```

After the header section, each file that is part of the project is analyzed. The code coverage results are displayed for each function, with references towards the C code:

```
1. File msc8156_main.c in D:/CW_builds/CW130/Training_SA/Test/Source/
    Last Modified: Sat Oct 16 12:53:56 2010
    Checksum: 5A78589

    File Summary (SC): 3 Functions
        9 blocks
            9 100.0%  covered
            0   0.0%  not covered
       38 statements
           38 100.0%  covered
            0   0.0%  not covered

    1.1 Function func1
        int func1()

        Function Summary (SC):
            5 blocks
                5 100.0%  covered
                0   0.0%  not covered
           28 statements
               28 100.0%  covered
                0   0.0%  not covered

        Block 1: 7 statements, lines 44 to 51
            { #pragma noinline Word16 YNM1=0, YNM2=0; Word32 T
            N,TNP1,YN,YNP1; int i; for (i = 0; i < DataBlockSi
            COVERED
```

An automated script can be implemented to parse this file and get other type of statistics useful for users like:

1. The number of functions within each file linked with application.
2. The used and unused functions.
3. A list of compiler built-in functions that the application uses.

6 Revision History

Table 1 provides a revision history for this application note.

Table 1. Revision History

Rev. Number	Date	Substantive Change
0	01/05/11	Initial release.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number:
Rev. 0
01/2011