Application Report
SPRA761 – June 2001

Techniques for Profiling on ROM-Based Applications

Harsh Sabikhi *Code Composer Studio, Applications Engineering*

ABSTRACT

This application report describes the methods for profiling on ROM and techniques for counting inclusive cycle counts. First, profiling will be described in some detail and then two examples will be given to demonstrate how to statistically view inclusive cycle counts on ROM.

Requirements

- Microsoft Windows 98, 2000 or NT
- TI Code Composer Studio™ for Microsoft Windows (version 2.0 or greater)

Contents

1 Introduction 1
2 Code Composer Studio Profiler in RAM 2
3 Code Composer Studio Profiler in ROM 2
3.1 Profiling in ROM Using Reset Clock Method 2
3.2 Profiling in ROM Using Cycle Count Difference Method 3
4 Profiling on ROM Example Using Reset Clock Method 3
4.1 Opening the Project and Creating the GEL File 3
4.2 Toggling Breakpoints/Probe Points and Connection to File I/O 3
4.3 Running the Application Connection to File I/O 4
5 Profiling on ROM Example Using Cycle-Count Difference Method 4
5.1 Opening the Project and Creating the GEL File 4
5.2 Toggling Breakpoints/Probe Points and Connection to File I/O 5
5.3 Running the Application Connection to File I/O 5
Appendix A: Source Code (GEL_CLKCpy.gel) 6
Appendix B: Source Code (GEL_CLKCpy2.gel) 7

1 Introduction

Profiling is a technique used to determine how much time a processor spends in a particular section of the code. The Code Composer Studio (CCS) profiler gives the developer some insight on how to identify sections of code that require optimization. A profile analysis can report how many cycles a particular function takes to execute and how often this function is called. This allows the developer to identify and eliminate performance barriers, thus creating a more efficient application. Often it is desired to profile code on ROM.

Code Composer Studio is a trademark of Texas Instruments.

All trademarks are the property of their respective owners.

When profiling on ROM, there are a limited number of hardware breakpoints available. This can limit the developers profiling range if the code contains numerous branches. Therefore, a more optimal workaround would be to limit the use of breakpoints/profile points. One way of achieving this is to use probe points instead of profile points. This process will be discussed in more detail in section 3.

2 Code Composer Studio Profiler in RAM

The CCS v2.0 profiler gives the developer a statistical view of the sections of code highlighted. It includes useful information such as code size, cycle counts, and average count. Once the developer has initialized start and end breakpoints, the profiler scans the piece of code and sets breakpoints at the start and end of each profile area, and also before and after every branch/function call. A profile point collects statistics on events that have occurred since the previous profile point was encountered. These are similar to breakpoints except they do not halt the processor. Profile points stop at breakpoints, collect the relevant data, and then resume compilation. Since the profiler is based on cycle counts or clock cycles, the more branches/function calls within the range of code being profiled, the slower profiling will be. Once the information of a particular branch is captured, the profile counter is accumulated. Profile counter counts cumulative and average cycles used in a specific range of code.

3 Code Composer Studio Profiler in ROM

As previously discussed in the introduction, there are some limitations with profiling in ROM. A workaround to this problem would be to profile before a branch, save the results, and then profile after the branch and save the results. Finally, adding all the results gives you the exclusive count for the total range. This solution can get tedious and time-consuming if the code has a large number of branches. An alternative to this procedure is to use probe points and connect them to file input/output. There are two ways of doing this, and they both will be discussed in the next two subsections.

3.1 Profiling in ROM Using Reset Clock Method

When there are a limited number of breakpoints available, one can still gain such statistics as cycle counts within large regions of code without the use of many breakpoints/profile points. Once the region is defined, the developer can place a probe point wherever there is a need to perform a specific task or function. For example, the probe point could be linked to a GEL file that records the value of the CLK register and writes it to another memory location. The value of the CLK register that is recorded is dependent on the starting and ending hardware breakpoint.

With the reset clock method, when the first hardware breakpoint is encountered at the beginning of the range, the CLK register is reset to zero (using a conditional probe point to call a GEL function). When the end hardware breakpoint is reached at the end of the range, the value of the CLK register is stored in memory. Finally, the last probe point is connected to the memory address written to, and this reads out the value to a file using File I/O. The File I/O feature uses the Probe Point concept, which allows you to extract/inject samples or take a snapshot of memory locations at a point you define. When the execution of the program reaches a probe point, the connected object (in this case a file) is updated.

Note: File I/O does not support real-time data transfer.

3.2 Profiling in ROM Using Cycle Count Difference Method

Rather than resetting the CLK register in the beginning, the developer may desire to keep the current value of the register. Now, when the initial breakpoint is encountered, the value of the CLK register must be stored in memory. When the profiler reaches the final breakpoint, the value of CLK is stored in another memory location. The reason why the CLK register value must be stored in memory is because if you wish to use file I/O to extract data to a file on your PC, the data must be obtained from the data page and not the register page. Finally, when the last probe point is reached, a specific GEL function can subtract the final value from the initial value, and this is the value of the cycle count that is read out to a file using file I/O.

Note: Make sure that the memory is mapped as ROM so that Code Composer Studio will use a hardware breakpoint-based Probe Point, instead of a software breakpoint.

4 Profiling on ROM Example Using Reset Clock Method

We now proceed in creating a test GEL function that will first reset the value of the CLK register and then copy the value of CLK into a specific memory location. For this illustration, the modem.pjt project is used.

Note: This illustration is done on a C5510 target with the modem.pjt. The free-memory addresses used to write the value of CLK vary with the target.

4.1 Opening the Project and Creating the GEL File

1. Launch Code Composer Studio.
2. Go to Project->Open and select <install directory>\tutorial\<target>\modem.
3. Select modem.pjt and open it.
4. Go to File->New->Source File. Create a GEL file called GEL_CLKCpy.gel and load this file by choosing File->Load GEL and selecting GEL_CLKCpy.gel. The source is provided in Appendix A.

4.2 Toggling Breakpoints/Probe Points and Connection to File I/O

1. Open modemtx.c from the project view, go to line 318 {Initialize ();}, and place a probe point and a breakpoint at that location.
2. Next place a probe point and a breakpoint on line 323 {g_ModemData.OutputBuffer[i] = SineLookup(i*g_ModemData.carrierFreq);}.
3. Go to Debug->Probe Points. Select modemtx.c line 318. In the probe type drop-down menu, choose Probe at Location if expression is TRUE. In the expression field, type in GEL_CLKReset () and click Replace. Recall that this function resets the CLK register.
4. Repeat step 3 for modemtx.c line 323, except in the expression field, type in GEL_CLKCpy() and click Replace. Close the dialogue by clicking OK.

5. Choose File->File I/O and click the File Output tab. Click add file. Call the data file clkcount.dat and click OK. Change the address field to any free-memory location (on the c5510 the free memory address is 0x6000 for the modem.pjt) and the length to 2. Now we are ready to add a probe point to this file.

6. Click Add probe points. Select modemtx.c line 323. In the probe type field, select Probe at Location, and in the connect to field, select FILE OUT: C:\..\modem\clkcount and click Add. Click OK, and then OK again to close the File I/O box.

4.3 Running the Application Connection to File I/O

1. From the toolbar menu, choose Rebuild All, and then load the program by choosing File->Load Program. Select modem.out and click OK.

2. Go to Profile->enable clock and then Profile->view clock.

3. In addition, you can open up a watch window to view the contents of CLK. Click on the watch window icon in the toolbar. In the watch 1 tab, under the Name field, type in CLK.

4. Choose Debug->Go Main and then Debug->Run. Notice the value of both CLK in the watch window and clock in the view clock window are 0.

5. Run to the next breakpoint by choosing Debug->Run. Now the value of clock is a number. This is the number that will be stored in memory and the file I/O is connected to this memory address.

6. To view the contents of your data file, choose File->Open and select clkcount.dat. Verify that the data is correct.

5 Profiling on ROM Example Using Cycle-Count Difference Method

This method is similar to the reset clock method, except now we do not reset the original clock value to 0. Now we create a GEL function, which first takes the value of CLK and stores it in memory. When the next breakpoint is encountered, the GEL function takes the new value of CLK and stores it in another memory location. Finally, the GEL function subtracts the new CLK value from the old one and connects it to File I/O.

5.1 Opening the Project and Creating the GEL File

1. Launch Code Composer Studio.

2. Go to Project->Open and select <install path>\tutorial\<target>\modem.

3. Select modem.pjt and open it.

4. Go to File->New->Source File. Create a GEL file called GEL_CLKCpy2.gel. Load this file by choosing File->Load GEL and selecting GEL_CLKCpy2.gel. The source is provided in Appendix B.

5.2 Toggling Breakpoints/Probe Points and Connection to File I/O

1. Open modemtx.c from the project view and go to line 318 {Initialize ();}. Place a probe point and a breakpoint at that location.
2. Next place a probe point and a breakpoint on line 323 {g_ModemData.OutputBuffer[i] = SineLookup(i*g_ModemData.carrierFreq);}.
3. Go to Debug->Probe Points. Select modemtx.c line 318. In the probe type drop-down menu, choose Probe at Location if expression is TRUE. In the expression field, type in GEL_CLKCpy () and click replace.
4. Repeat step 3 for modemtx.c line 323 except in the expression field, then type in GEL_CLKCpy2() and click replace. Close the dialogue by clicking ok.
5. Choose File->File I/O and click the File Output tab. Click Add File. Call the data file clkcount2.dat and click ok. Change the address field to any free memory location (on the c5510 the free memory address is 0x6000 for the modem.pjt) and the length to 2. Now we are ready to add a probe point to this file.

   Note: This above address is valid for the c5510. Please modify the address accordingly to your target.
6. Click Add Probe Points. Select modemtx.c line 323. In the Probe Type field, select Probe at Location. In the Connect To field, select FILE OUT: C:\..\modem\clkcount2 and click Add. Click OK and then OK again, to close the File I/O box.

5.3 Running the Application Connection to File I/O

1. From the toolbar menu, choose Rebuild All, and then load the program by choosing File->Load Program. Select modem.out and click OK.
2. Go to Profile->enable clock and then Profile->view clock.
3. In addition, you can open up a watch window to view the contents of CLK. Click on the watch window icon in the toolbar. In the watch 1 tab, under the Name field, type in CLK.
4. Choose Debug->Go Main and then Debug->Run. Notice now that the CLK value is incremented from its original value.
5. Run to the next breakpoint by choosing Debug->Run. Now the value of CLK is another number. The GEL function takes the difference of the two CLK values and writes it to another free memory location, which will then be written to the host using File I/O.
6. To view the contents of your data file, choose File->Open and select clkcount2.dat. Verify that the data is correct.

Appendix A: Source Code (GEL_CLKCpy.gel)

```
menuitem "GEL Clock Counter";
hotmenu GEL_CLKCpy()
{
long *address;
address = 0x6000@data;        /* free memory on the C5510 */
*address = (long) CLK;
}
hotmenu GEL_CLKReset()
{
CLK = 0;
}
```

NOTE: the value of “address” can vary, depending on the target used.

Appendix B: Source Code (GEL_CLKCpy2.gel)

```
menuitem "GEL Clock Counter";

hotmenu GEL_CLKCpy()
{
long *address;
address = 0x6000@data;        /* free memory on the C5510 */
*address = (long) CLK;
}

hotmenu GEL_CLKCpy2()
{
long *address2, *final_value, *address;
address = 0x6000@data;
address2 = 0x6004@data;       /* free memory on the C5510 */
*address2 = (long) CLK;

final_value = 0x6008@data;    /* free memory on the C5510 */
*final_value = *address2 - *address;
}
```

NOTE: the value of “address” can vary, depending on the target used.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2001, Texas Instruments Incorporated