Analysis Toolkit v1.3 for Code Composer Studio ™ User's Guide

TEXAS
INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI 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 and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments

Post Office Box 655303 Dallas, Texas 75265

Read This First

About This Manual

The Analysis Toolkit (ATK) is a tool that helps enhance the robustness and analyze the performace of embedded DSP applications. The toolkit provides visualization of source line coverage information, which helps you to construct tests to ensure adequate coverage of your code. It also provides exclusive profile data for individual functions and source line profile data for individual lines of code for each function.

This user's guide describes how to set up the ATK, how to use the ATK for data collection, and how to visualize data.

Notational Conventions

This document uses the following conventions.

❑ Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **bold version** of the special typeface for emphasis; interactive displays use a **bold version** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

Here is a sample program listing:

```
0011  0005  0001          .field    1, 2
0012  0005  0003          .field    3, 4
0013  0005  0006          .field    6, 3
0014  0006                .even
```

Here is an example of a system prompt and a command that you might enter:

```
C:  csr -a /user/ti/simuboard/utilities
```

Trademarks

Trademarks are the property of their respective owners.

Contents

Figures

Tables

Introduction

This chapter introduces the Analysis Toolkit (ATK) code coverage and exclusive profile tool. The code coverage and exclusive profile tool is supported on all the C5500 and C6000 simulators in Code Composer Studio™ (CCStudio™) version 3.1.

1.1 Introduction to the ATK

The Analysis ToolKit (ATK) code coverage and exclusive profile tool helps you create robust application software and analyze the performance of the application.

The code coverage information allows you to assess which lines of code were used in a run of the software. This information collated across multiple runs for the application with different test vectors can help indicate the overall line coverage achieved and can help you focus on creating specific test vectors to cover the lines not covered so far.

The exclusive profile information displays the behavior of the functions through various measurements, such as:

❏ Number of target instructions corresponding to the function

❏ Number of cycles taken by the function

❏ Number of stalls caused by this function

❏ Other events, depending on the program target

This exclusive profile data also splits up each of these measurements by the source lines of the functions. Using this profiler information, you can determine which sections of the code are working inefficiently, and which sections to focus on for the other areas.

For example, the instruction count data and the cycle data for a given function measured on the C6000 simulator can help indicate the instructions per cycle achieved for this function. This information can be used to understand what amount of parallelism of the target architecture is being exploited by this function and in turn can help assess if you need to focus on increasing the parallelism. The source line view of this data for the same function can help indicate possible pieces of code in this function to focus on.

The code coverage and exclusive profile data is displayed in a Microsoft Excel™ file, with multiple worksheets. The function level data from a given run is captured in a summary worksheet. The data for each of the source files corresponding to the current program is captured in separate worksheets. The summary sheet has cross–links from the function entries to the corresponding detailed data in a different sheet, making it easy to browse through the information.

Appendix A contains information you should be aware of before beginning to use the ATK.

1.2 How the ATK Fits in the Software Development Cycle

Typically, development starts with the creation of algorithms. At this stage, use the code coverage tool for thorough validation of algorithms. Subsequently, algorithms are tuned for high performance, meeting performance budgets. Similarly, the exclusive profiler may be used to identify the location of performance losses and their contributing factors.

See Figure 1−1 for a quick summary of how the ATK fits into the development cycle.

Figure 1−1. Using the ATK Tools for Algorithm and Application Development
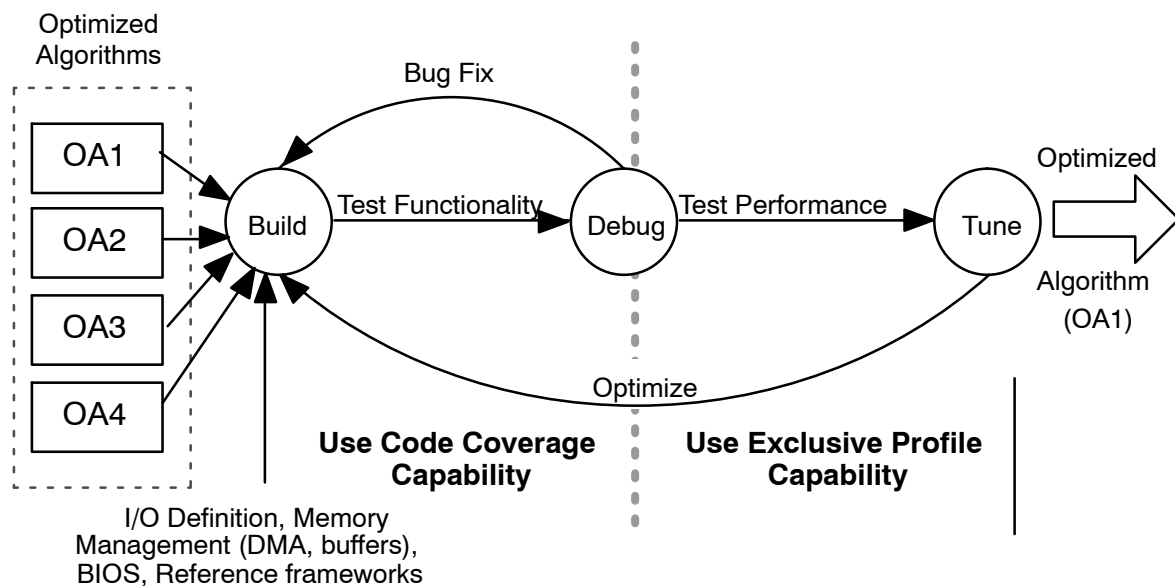
Getting Started with the ATK

This chapter provides instructions for getting started, see Appendix A for more valuable information.

2.1   Accessing and Using the ATK

After installation, the ATK is accessible from the CCStudio menus (see Figure 2−1 and Figure 2−5). You will need to select the appropriate simulator configuration to use the ATK.

After selecting the appropriate simulator in CCStudio Setup, adding it, and saving it, the analysis toolkit software is ready for use.

Figure 2−1. ATK Menu Items
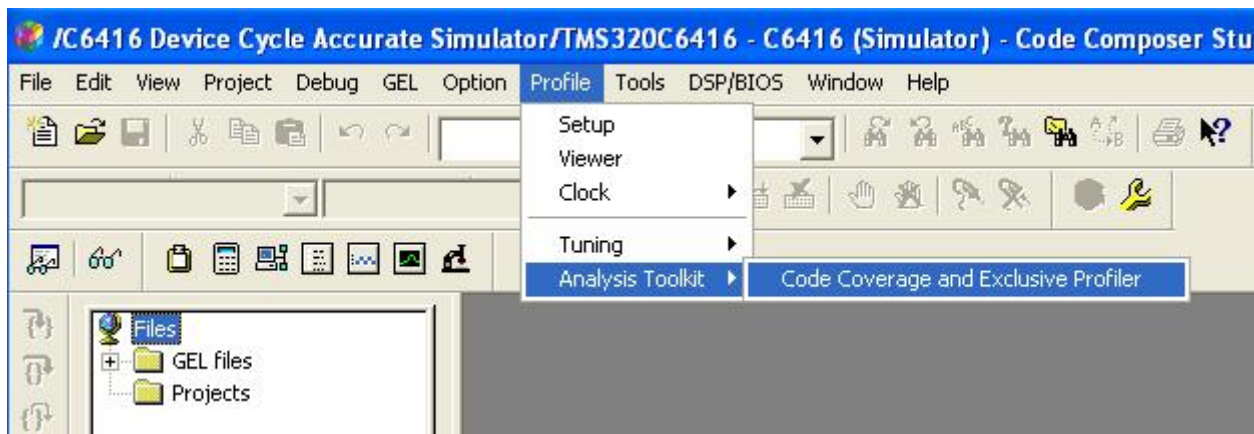
2.2   Setting Up the Simulator

You will not be able to access the ATK until you have set up the appropriate simulator as follows:

1) Launch Setup by clicking on the Setup CCStudio v3.1 icon on your desktop or from within the CCStudio by choosing Launch Setup from the File menu.

2) Select the correct simulator from the Available Factory Boards list. In this example, the C6416 Device Cycle Accurate Simulator is selected. This is the simulator required to run the example (tree.pjt) used in section 2.3 on the C6000 platform. However, the ATK works with all the simulators available on the C5500 and C6000 platforms.

Figure 2−2. CCStudio Setup Screen
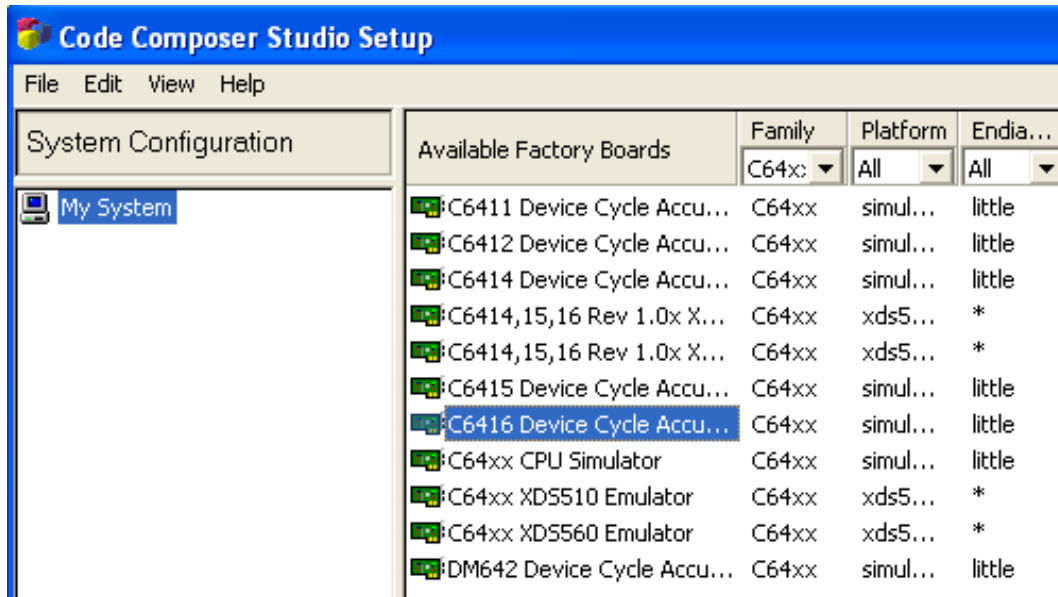
3) Add the configuration, and save and quit CCStudio Setup.

2.3 Taking a Quick Tour

This section will take you on a quick tour that will help you get started with the code coverage and exclusive profile tool. In this tour, we use the Tree example that is packaged as a part of code coverage and exclusive profile installation.

1) Launch Setup by clicking on the Setup CCStudio v3.1 icon on your desktop or from within the CCStudio by choosing Launch Setup from the File menu.

2) Select the correct simulator from the Available Factory Boards list.

■ If you are working on the 5500 platform, select the C55xx Cycle Accurate Simulator.

■ If you are working on the C6000 platform, select the C6416 Device Cycle Accurate Simulator.

Figure 2−3. CCStudio Setup Screen
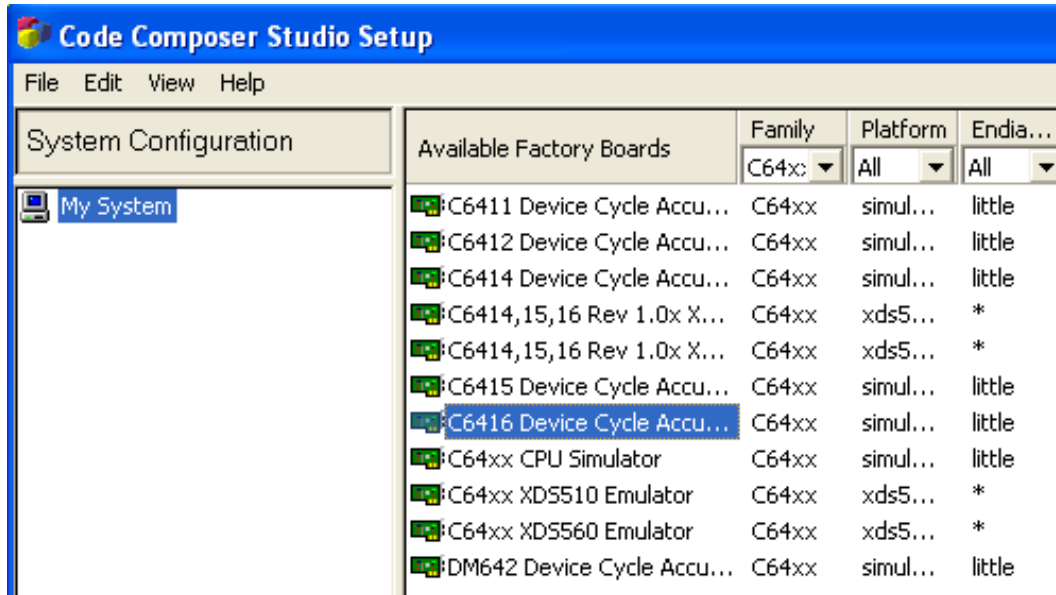
3) Add the configuration, and save and quit Code Composer Studio Setup.

4) Launch CCStudio.

5) Select Project→Open, and browse to the project file tree.pjt.

   ■ If you are working on the 5500 platform, it will be <CCS_INSTALL_DIR>:\examples\sim55xx\code_coverage\tree\ tree.pjt.

   ■ If you are working on the C6000 platform, it will be <CCS_INSTALL_DIR>:\examples\sim64xx\code_coverage\tree\ tree.pjt.

6) Click Open.

7) Select Project→Build Options. Verify that the compiler command at the top of the window includes the −g option, i.e., with full symbolic debug information. The other options may vary depending on the DSP board you are using. See the CCStudio online help for more information about the compile file build options.

8) Select Project→Rebuild All or click the Rebuild All toolbar button.

9) Choose File→Load Program. Select the program you just rebuilt, tree.out, in the Debug folder and click Open.

10) Select Profile Setup from the CCStudio Profile menu, shown in Figure 2−4.

11) Click on the Enable/Disable Profiling button and select Collect Code Coverage and Exclusive Profile Data in the Profile Setup window shown in Figure 2−5.

Note:

When profiling is enabled, a clock will appear on the bottom of the CCStudio window.

Figure 2−4. Profile Setup Menu Option
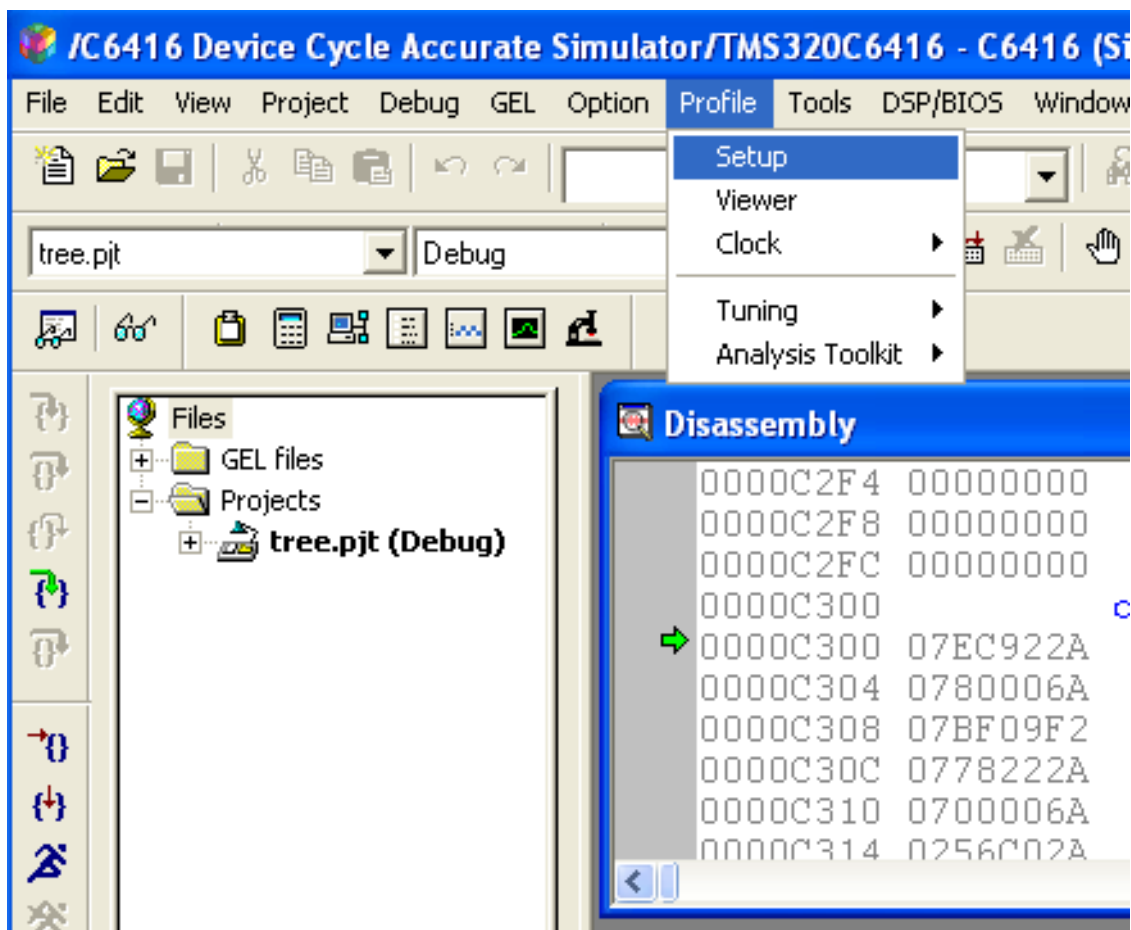
Figure 2−5. Profile Setup Screen
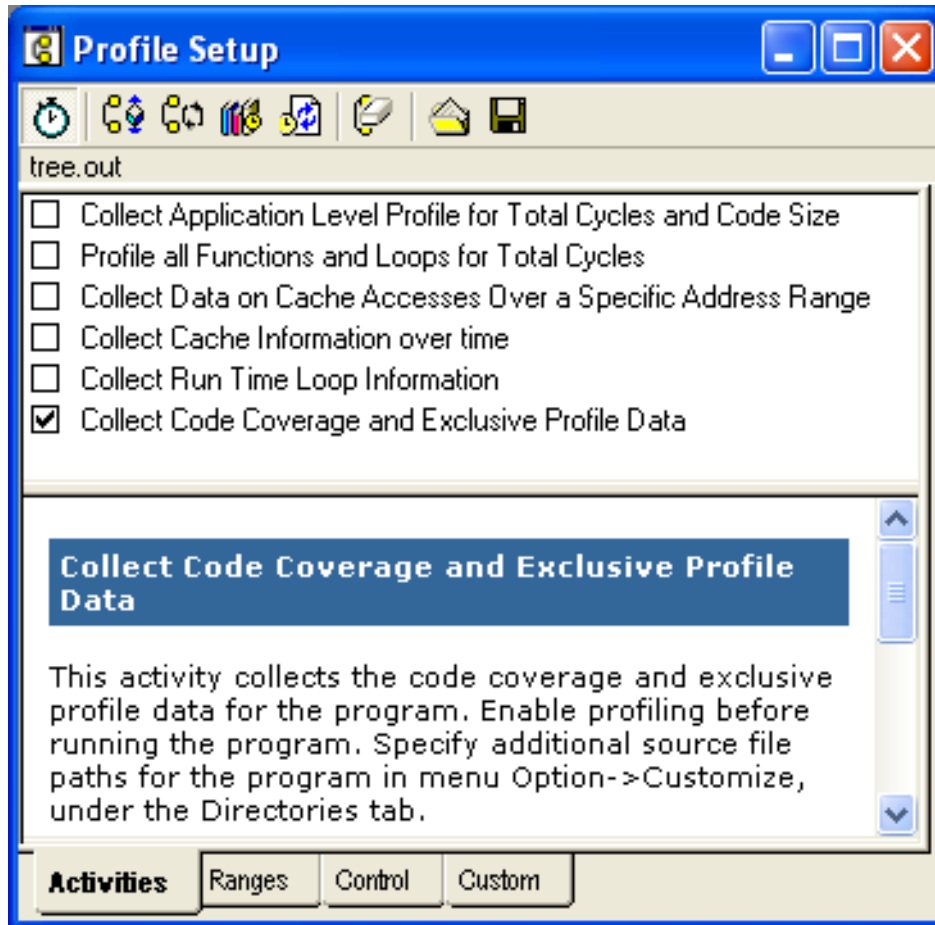
12) Execute your program by clicking on the toolbar Run button or by selecting Debug→Run.

13) Select Profile→Analysis Toolkit→Code Coverage and Exclusive Profiler to view the coverage and exclusive profile data. It will take a few minutes to appear, as data processing occurs at this point.

14) A Microsoft Excel screen such as that shown in Figure 2−6 will appear, displaying the function level summary information.

Figure 2–6. Code Coverage and Exclusive Profiler Summary Data Worksheet

	A	B	C	D	E	F	I	K	L	M	N
	Function	File	Line no.	Size(bytes)	Start address(hex)	#times called	%coverage	Total Instructions	cycle.Total	cycle.CPU	L1P.miss. summary
2	inOrder	tree.c	126	164	0x000039a8	16	80	497	1222	1108	40
3	isnum	tree.c	25	244	0x00003620	16	73	880	2292	2032	72
4	main	main.c	36	696	0x00006740	1	88	731	1679	1257	31
5	postOrder	tree.c	109	160	0x00003908	16	80	490	1109	1108	12
6	preOrder	tree.c	91	164	0x00003864	16	80	497	1160	1140	24
7	treeInsert	tree.c	44	336	0x00003714	16	100	1672	3880	3619	158
8	Others							154303	1098652	134463	4179
9	Total							159070	1109994	144727	4516

15) You can click on the function name to view the source line information, illustrated in Figure 2–7.

Figure 2–7. Code Coverage and Exclusive Profiler Source Line Information Worksheet

	A	B	C	D	E	F	G	H
1	Line Count: Min	Line Count: Max	Line Number	Source Line	Total Instructions	cycle.Total	cycle.CPU	L1P.miss. summary
110	16	16	109	void postOrder(struct Node * root){	64	96	96	0
111	16	16	110	if (root == NULL){	32	113	112	0
112	0	0	111	printf("Empty tree...Please sow	0	0	0	2
113	0	0	112	return;	0	0	0	0
114			113	}				
115			114	else{				
116	9	16	115	if (root->left != NULL){	64	192	192	2
117	7	7	116	postOrder(root->left);	21	42	42	0
118			117	}				
119	9	16	118	if (root->right != NULL){	80	256	256	2
120	7	7	119	postOrder(root->right);	21	42	42	0
121			120	}				
122	16	16	121	printf("%d ",root->data);	160	192	192	2
123			122	}				
124	16	16	123	}	48	176	176	4

2.4 Interpreting the Output Information

The tool brings up the summary worksheet in Excel, shown in Figure 2–8. This sheet shows code coverage and exclusive profile information for all the functions in the program. In addition, it shows two rows (Others and Total) after the functions summary. The total of all the event counts for the entire application is shown in the Total row. The counts incurred in the portion of the code with no debug information are shown in the Others row. It is possible to expand each of these functions to view the source level information.

Table 2–1 describes the data shown in Figure 2–8. Table 2–2 describes the C6000-specific events, while Table 2–3 describes C5500-specific events.

Figure 2–8. Code Coverage and Exclusive Profiler Summary Data Worksheet

	A	B	C	D	E	F	I	K	L	M	N
1	Function	File	Line no.	Size(bytes)	Start address(hex)	#times called	%coverage	Total Instructions	cycle.Total	cycle.CPU	L1P.miss. summary
2	inOrder	tree.c	126	164	0x000039a6	16	80	497	1222	1108	40
3	isnum	tree.c	25	244	0x00003620	16	73	880	2292	2032	72
4	main	main.c	36	696	0x00006740	1	88	731	1679	1257	31
5	postOrder	tree.c	109	160	0x00003908	16	80	490	1109	1108	12
6	preOrder	tree.c	91	164	0x00003864	16	80	497	1160	1140	24
7	treeInsert	tree.c	44	336	0x00003714	16	100	1672	3880	3619	158
8	Others							154303	1098652	134463	4179
9	Total							159070	1109994	144727	4516

Table 2–1. Common Data Shown in Summary View of Functions Coverage and Exclusive Profile for C5500 and C6000 Platforms

Column Number	Column Name	Description
1	Functions	Lists all the function symbols in the program.
2	File	Names the source file that has the function.
3	Line No.	Provides the line number in the file where the function begins.
4	Size (bytes)	Provides the size of the function in bytes.
5	Start Address (hex)	Provides the function start address in hexadecimal.
6	# times called	Lists the number of times the function was called during the execution of the program.
7	% coverage	Shows the number of lines executed/number of lines of code as % coverage.
8	Total Instructions	Shows the cumulative count of the number of instructions decoded for that function.

Table 2–2. Event Annotations for C6000 Platforms

Column Number	Column Name	Description
9	cycle.Total	Available only on C6x1x Device Cycle Accurate simulators. Not applicable for CPU and Device Functional simulators.
10	Cycle.CPU	
11	L1P.miss.summary	Columns 11 to 21 are available only on C6x1x simulators.
12	L1P.miss.summary_rate	L1P cache miss rates for read, expressed as a percentage of total L1P accesses.
13	L1D.miss.read	
14	L1D.miss.read_rate	L1D cache miss rates for read, expressed as a percentage of total L1D reads.
15	L1D.miss.write	
16	L1D.miss.write_rate	L1D cache miss rates for writes, expressed as a percentage of total L1D writes.
17	L1P.hit	
18	L1D.hit.read	
19	L1D.hit.write	
20	L1P.miss.conflict	
21	L1D.miss.conflict	
22	CPU.instruction.condition_false	

For information on these events, see the CCStudio online help.

Table 2−3. Event Annotations for C5500 Platforms

Column Number	Column Name	Description
9	CPU.execute_packet	
10	cycle.CPU or cycle.Total	This column is not available for the C55xx Functional simulators.
		❏ cycle.CPU is displayed for the C55xx Cycle Accurate simulator.
		❏ cycle.Total is displayed for the C5510 and C5502 Device simulators.
11	CPU.stall.prefetch	Not available for the C55xx Functional simulator.
12	CPU.stall.ppu.summary	−do
13	CPU.stall.mem.read	−do
14	CPU.stall.mem.write	−do
15	CPU.access.data.read	
16	CPU.access.data.write	
17	Program_cache.hit	Available only for the C5510 and C5502 Device simulators.
18	Program_cache.miss	−do
19	CPU.instruction.condition_false	

For information on these events, see the CCStudio online help.

When you click on the function name in the summary view, the worksheet corresponding to the source file that function belongs to is brought to the front; see Figure 2−9. This worksheet shows the expanded source level view of that function.

Figure 2−9. Expanded Source Level View of Function

tree_summary.xls

	A	B	C	D	E	F	G	H
1	Line Count: Min	Line Count: Max	Line Number	Source Line	Total Instructions	cycle.Total	cycle.CPU	L1P.miss. summary
110	16	16	109	void postOrder(struct Node * root){	64	96	96	0
111	16	16	110	if (root == NULL){	32	113	112	0
112	0	0	111	printf("Empty tree...Please sow	0	0	0	2
113	0	0	112	return;	0	0	0	0
114			113	}				
115			114	else{				
116	9	16	115	if (root->left != NULL){	64	192	192	2
117	7	7	116	postOrder(root->left);	21	42	42	0
118			117	}				
119	9	16	118	if (root->right != NULL){	80	256	256	2
120	7	7	119	postOrder(root->right);	21	42	42	0
121			120	}				
122	16	16	121	printf("%d ",root->data);	160	192	192	2
123			122	}				
124	16	16	123	}	48	176	176	4

The coverage data for the source line view shows the data described in Table 2–4:

Table 2–4. Data Shown in the Expanded Source Level View of the Function

Column Number	Column Name	Description
1	Line Count: Min	Shows the minimum of the execution counts of any of the assembly instructions corresponding to this Source Line. A zero here indicates that there is at least one assembly instruction corresponding to this source line that has not been executed.
2	Line Count: Max	Shows the maximum of the execution counts of any of the assembly instructions corresponding to this source line. A 0 here indicates that none of the assembly instructions corresponding to this source line have been executed.
		When a source file is compiled, each line in the source file is actually converted into 0 or more instructions. Consider a particular source line. Let us say, for example, that 3 instructions are generated for this line. When the program is executed, suppose that the first instruction corresponding to this line executed twice, the second instruction executed once and the third instruction was never executed. Thus, the Line Count:Min for this source line would be the minimum of (2, 1, 0), which is 0. The Line Count:Max for this source line would be the maximum of (2, 1, 0), which is 2.
3	Line Number	Provides line number in the file where the source is
4	Source Line	Lists the content of the source line in the file
5	Total Instructions	The cumulative count of the total number of the target instructions (corresponding to the source line) that were decoded during the run of the program.

2.5 Understanding the Color Coding

The summary view shows the code coverage data per function. The colors indicate the following:

❑ Red: The functions that were not covered (0 in the %Coverage column) are shown in red.

❑ Green: The functions that have coverage (any number greater than 0 in the %Coverage column) are shown in green.

❑ Black: Comments and the Others and Total rows are shown in black.

The function detailed view shows the source level view of the function chosen. The colors indicate the following:

❑ Red: The lines that were not covered (0 in the Total Instructions column) are shown in red.

❑ Green: The lines that were covered (any number greater than 0 in the Total Instructions column) are shown in green.

❑ Black: Comments and the source lines for which no code is generated are shown in black.

Additional Methods of Collecting Code Coverage and Exclusive Profile Data

This chapter helps you use the code coverage and exclusive profile tool for different situations and different purposes. Appendix B provides the specific application programming interface (API) information and code examples you need to carry out these steps. Appendix C contains examples, and Appendix D lists the trace files generated and their contents.

3.1 Entering Additional Source File Paths

The coverage and exclusive profile tool requires source code information to generate code coverage data. If you have a CCStudio project–based flow, where all the source files are present as a part of the project, then the coverage tool picks up the source information from the project. However, if you have a make file–based flow or static libraries as a part of the project, then you must specify the additional source file paths that contain the sources that formed the executable. You can do so by selecting menu options Option→Customize→Directories to get to the screen shown in Figure 3–1.

Figure 3–1. Screen to Enter Additional Source File Paths
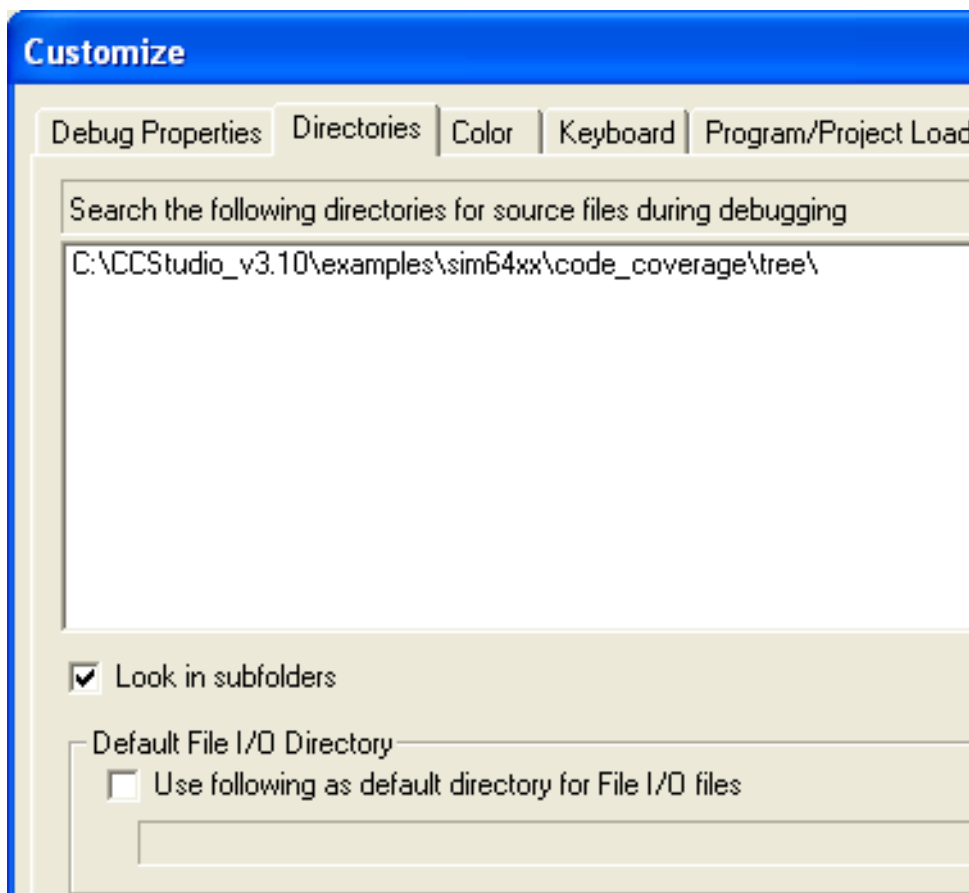
3.2   Collecting Data for Continuous Programs

The majority of DSP applications run in an infinite loop, processing the input data and producing the output data. You can also collect code coverage and exclusive profile data for this type of applications. The sequence of steps is as follows:

1)   If you have not yet set up CCStudio for your simulator, launch Setup by clicking on the Setup CCStudio v3.1 icon on your desktop or from within the CCStudio by choosing Launch Setup from the File menu.

2)   Select the appropriate simulator from the Available Factory Boards list, add, save, and quit.

3)   Launch CCStudio if it is not already running.

4)   Open the project file.

5)   Select Project→Build Options. Verify that the compiler command at the top of the window includes the −g option, i.e., with full symbolic debug information. The other options may vary depending on the DSP board you are using.

6)   Select Project→Rebuild All or click the Rebuild All toolbar button.

7)   Choose File→Load Program. Select the program you just rebuilt, and click Open.

8)   Select Profile Setup from the CCStudio menu, shown in Figure 2−4.

9)   Enter additional source file paths (see section 3.1).

10) Click on Profile→Setup.

11) Enable profiling and select Collect Code Coverage and Exclusive Profile Data in the Profile→Setup window shown in Figure 3−2.

12) Set the breakpoint in the function where you want to stop collecting coverage and exclusive profile data (see CCStudio online help for information on how to insert a breakpoint).

13) Run the program until the breakpoint is hit.

14) Disable profiling by clicking on the Enable/Disable Profiling button (stop clock icon) shown in Figure 3−2.

Figure 3−2. Enable/Disable Profiling Button
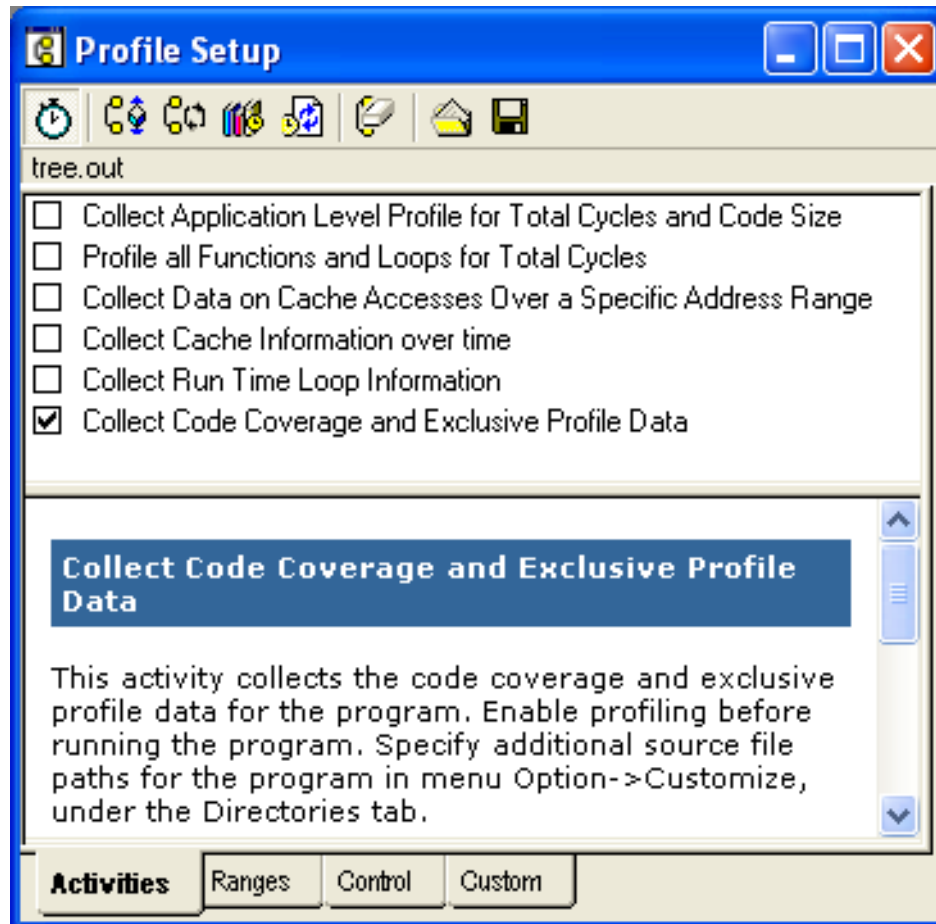
Click on Profile→Analysis Toolkit→Code Coverage and Exclusive Profiler to view the code coverage and exclusive profile data.

3.3 Collecting Data between Two Points in a Program

You can collect the coverage and exclusive profile data between two points in program execution. This approach is very similar to the approach in the previous section, except that you must set a breakpoint at the point you want coverage and exclusive profile data collection to start and then enable profiling. When the program reaches the breakpoint where you want data collection to stop, you must disable profiling. The sequence of steps will be as follows:

1) If you have not yet set up CCStudio for your simulator, launch Setup by clicking on the Setup CCStudio v3.1 icon on your desktop or from within the CCStudio by choosing Launch Setup from the File menu.

2) Select the appropriate simulator from the Available Factory Boards list, add, save, and quit.

3) Launch CCStudio if it is not already running.

4) Open the project file.

5) Select Project→Build Options. Verify that the compiler command at the top of the window includes the –g option, i.e., with full symbolic debug information. The other options may vary depending on the DSP board you are using.

6) Select Project→Rebuild All or click the Rebuild All toolbar button.

7) Choose File→Load Program. Select the program you just rebuilt, and click Open.

8) Set the breakpoints in the function where you want to start and stop collecting coverage and exclusive profile data (see CCStudio online help for information on how to insert a breakpoint).

9) Select Profile Setup from the CCStudio menu, shown in Figure 2–4.

10) Enter additional source file paths (see section 3.1).

11) Click on Profile→Setup.

12) Run the program until the breakpoint is hit.

13) Enable profiling and select Collect Code Coverage and Exclusive Profile Data in the Profile→Setup window shown in Figure 3–2.

14) Run the program until the other breakpoint is hit.

15) Disable data collection in the Profile→Setup window shown in Figure 3–2.

16) Click on Profile→Analysis Toolkit→Code Coverage and Exclusive Profiler to view the code coverage and exclusive profile data.

3.4 Collecting Inclusive Data for Selected Functions

It is possible to collect code coverage and exclusive profile data for particular functions for the complete run of a program. You can do this by using halt and resume points in the program. The sequence of steps will be as follows:

1) If you have not yet set up CCStudio for your simulator, launch Setup by clicking on the Setup CCStudio v3.1 icon on your desktop or from within the CCStudio by choosing Launch Setup from the File menu.

2) Select the appropriate simulator from the Available Factory Boards list, add, save, and quit.

3) Launch CCStudio if it is not already running.

4) Enter additional source file paths (see section 3.1).

5) Select Profile Setup from the CCStudio menu, shown in Figure 2−4.

6) Click on Profile→Setup.

7) Enable profiling and select Collect Code Coverage and Exclusive Profile Data in the Profile→Setup window shown in Figure 3−2.

8) Open the project file.

9) Set up a halt point at the beginning of the program by clicking and dragging the entry point to the Halt/Resume Collection section of the Control tab in the Profile Setup window, as shown in Figure 3−3.

Figure 3−3. Control Tab in Profile Setup Window
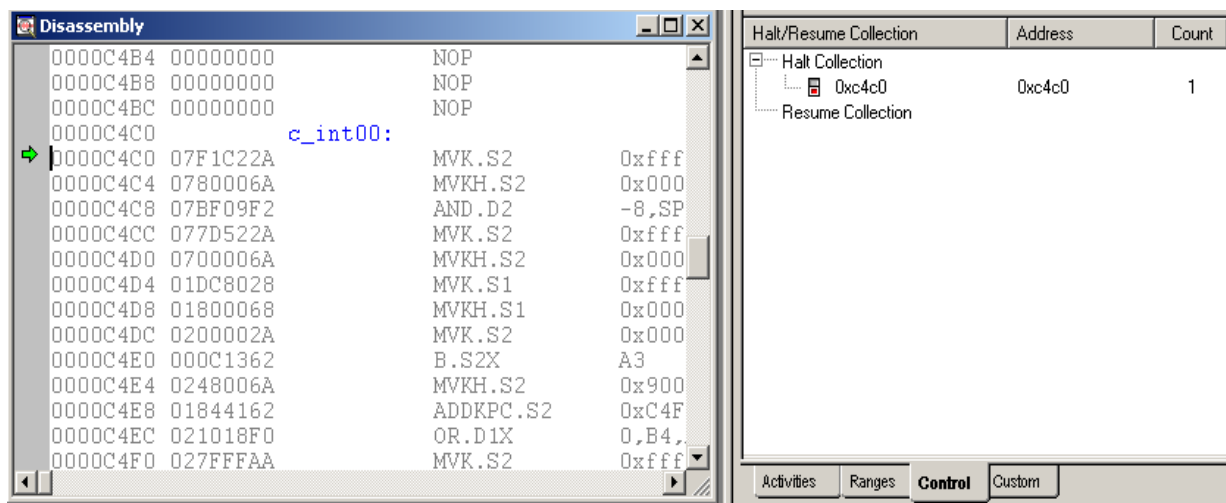
10) Now set up Resume and Halt collection points at the beginning and end of the function, as shown in function preOrder in the example shown in Figure 3−4.

Figure 3−4. Example of Halt and Resume Points in a Program
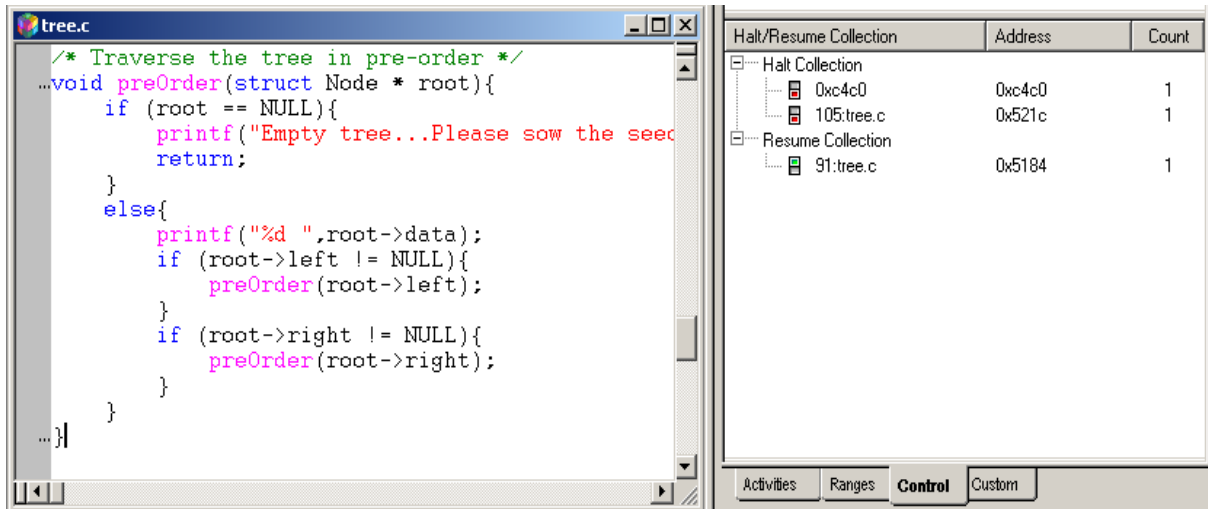
11) Run the program to completion (Debug→Run).

12) Click on Profile→Analysis Toolkit→Code Coverage and Exclusive Profiler to view the code coverage and exclusive profile data.

3.5 Collecting Data in Batch Mode

You may wish to automate the collection of coverage and exclusive profile data. Typically, if you want to collect coverage and exclusive profile data over many runs of the application or over many applications, then you should write a custom script by using the scripting APIs to the CCStudio. In Appendix B, we provide some sample APIs to perform coverage data collection through the script.

Typically, CCStudio Tuning scripting is installed under the folder:

<CCS Installation Path>\bin\utilities\ccs_scripting

The functions offered by the sample APIs include launching CCStudio, opening projects, and loading and running programs.

The simulator, when configured to collect code coverage and exclusive profile data, produces a file called <program_name>.tprof after the program execution. This tprof file contains the various target events binned against the corresponding program addresses. The code coverage and exclusive profile tool then uses the information present in the program file, the tprof file, and the sources corresponding to the program, to produce the coverage and exclusive profile data with source level information for the program.

The Perl module ATK_functions.pm, located in the CCStudio scripting directory, provides the data collection and post–processing functions.

3.5.1 Algorithm for Collecting Data in Batch Mode

This is the rough algorithm your script needs to follow to collect coverage and exclusive profile data. After reviewing these steps, you can then look for the APIs that help you to implement these steps along with any desired customizations.

1) Launch CCStudio.

2) Load the program.

3) Configure CCStudio to collect coverage data.

4) Enable coverage and exclusive profile data collection.

5) Run the program.

6) Disable coverage and exclusive profile data collection.

7) Post process the collected data.

Appendix B provides the specific application programming interface (API) information and code examples you need to carry out these steps within the script. Appendix C contains examples, and Appendix D lists the trace files generated and their contents.

3.5.2   Workflow for Cumulating Code Coverage Data

You may also wish to merge different coverage data collected from multiple runs of the application code. Typically, you may run the application many times, each time with a different set of test vectors. At the end of the runs, you may need to visualize the cumulative coverage obtained for the code over all the runs.

To merge coverage data, a function called ATKMergeCoverage is provided in the Perl module ATK_functions.pm, placed in the <CCS_INSTALL_DIR>\bin \utilities\ccs_scripting directory when the ATK package is installed. This function merges the tprof trace data collected in the current run with those already collected from previous runs.

The following steps are used to collect cumulative coverage data. CCStudio Tuning scripting–based Perl script needs to be written for achieving the following functionalities. Appendix C contains the APIs and example script.

Repeat steps 1–6 for all the test input files for a program loaded.

1)   Get the current test input file.

2)   Load the profile configuration to enable code coverage activity.

    a)   Select Collect Coverage Data.

    b)   Enable profiling by clicking on the Enable/Disable Profiling button.

3)   Run the program for inputs in the current test input file.

4)   Disable profiling after the program halts.

5)   Merge the coverage data (tprof file data) obtained in the current run with the already collected data from the previous runs. Use the ATKMergeCoverage function to achieve this.

6)   Process the cumulative coverage data collected over all the runs and generate the Excel spreadsheet.

Information to Be Aware Of

1) Trace files can grow to large sizes if analysis data is collected over long periods of simulation. Be sure to remove unwanted trace files. ***The ATK does not remove trace files.***

2) As mentioned earlier, whenever data collection is started, the associated trace files are reopened for writing; thus, any old contents are lost. To prevent loss of existing trace data,

    a) Either rename old trace files before program load, or

    b) Rename the program to be loaded

3) Typically, the visualization tools are used as part of an *analyze, modify, rebuild,* and *rerun* sequence for iterative analysis. This involves generating trace files with the same names as those already generated. Thus, before starting data collection for a subsequent run, visualization tools that are visualizing the trace output of the current run need to be quit. Alternatively, the trace files from the current run may be "saved as" copies and the copies visualized. Saving a copy also enables comparison of data from different runs.

APIs for Batch Mode Coverage and Exclusive Profile Data Collection

This section explains the application programming interfaces (APIs) required to control collection and post−processing of coverage data.

B.1 API for Configuring Coverage Data Collection

The API for configuring coverage data collection is named:
ATKConfigureCoverage(($ccs_scripting_object_ptr , $ccs_install_dir).

❑ Description: Configures coverage data collection in CCS

❑ Arguments:

■ CCS_SCRIPTING_PERL object pointer

■ CCS Installation directory

❑ Return value:

■ 0 on success

■ !=0 otherwise

B.1.1 Example

```
use CCS_SCRIPTING_PERL;

use ATK_functions;

my $MyCCScripting = new CCS_SCRIPTING_PERL::CCS_Script-
ing();

$CCS_DIR = "D:\\ti_c6000";

#Invoke CCS

...

#Load the program

...

#configure coverage

ATKConfigureCoverage($MyCCScripting, $CCS_DIR);
```

B.2 APIs for Controlling Data Collection

There are two APIs to control data collection: one to start the collection and one to stop it.

B.2.1   *Start Data Collection*

The API for starting data collection is named:
ATKStartDataCollection($ccs_scripting_object_ptr)

❑   Description: Enables profile data collection on the target simulator

❑   Arguments: CCS_SCRIPTING_PERL object pointer

❑   Return Value: Always 0

B.2.1.1 *Stop Data Collection*

The API to stop data collection is named:
ATKStopDataCollection($ccs_scripting_object_ptr).

❑   Description: Stops data collection on the target simulator and creates the trace file required for coverage processing

❑   Arguments: CCS_SCRIPTING_PERL object pointer

❑   Return Value: Always 0

B.2.1.2 *Example*

```
use CCS_SCRIPTING_PERL;
use ATK_functions;
my $MyCCScripting = new CCS_SCRIPTING_PERL::CCS_Script-
ing();
$CCS_DIR = "D:\\ti_c6000";
#Invoke CCS
...
#Load the program
...
#configure coverage
ATKConfigureCoverage($MyCCScripting, $CCS_DIR);
#Enable Data Collection
ATKStartDataCollection($MyCCScripting);
#Run the program
...
#Disable Data Collection
ATKStopDataCollection($MyCCScripting);
```

B.3 API for Generating Coverage Data File

The API to generate the coverage data file is named:
ATKTprof2xls($ccs_install_dir, $prog_file, $tprof_file, $source_paths)

❑ Description: Generates coverage data in files and Excel sheet from the tprof file data.

❑ Arguments:

■ CCS installation directory

■ Program file name

■ tprof file name

■ Semicolon–separated search path for program source files

❑ Return Value:

■ 0 on success

■ !=0 otherwise

B.3.1 Example

```
use CCS_SCRIPTING_PERL;
use ATK_functions;
my $MyCCScripting = new CCS_SCRIPTING_PERL::CCS_Script-
ing();
$CCS_DIR = "D:\\ti_c6000";
#Invoke CCS
…
#Load the program
…
#configure coverage
ATKConfigureCoverage($MyCCScripting, $CCS_DIR);
#Enable Data Collection
ATKStartDataCollection($MyCCScripting);
#Run the program
…
#Disable Data Collection
ATKStopDataCollection($MyCCScripting);
#Trace data(Tprof file) is available at this time in the
#directory of the "program.out" with the name "pro-
gram.tprof"
#Process the trace data and convert it to excel format
ATKTprof2xls($CCS_DIR, $MyProgram, $tprof, $src_path);
```

B.4 API for Merging Coverage Trace Files

The API to merge the coverage data trace files is named:
ATKMergeCoverage($ccs_install_dir, $tprof_curr,
$tprof_cumulated _till_now)

❏ Description – Merges coverage data across multiple runs of the program
this works at a tprof level

❏ Arguments:

■ CCS installation directory

■ tprof file of current run

■ tprof file that contains the cumulated data into which the current run's
tprof data needs to be merged

B.4.1 Example

```
use CCS_SCRIPTING_PERL;

use ATK_functions;

my $MyCCScripting = new CCS_SCRIPTING_PERL::CCS_Script-
ing();

$CCS_DIR = "D:\\ti_c6000";

#Invoke CCS

...

#Load the program

...

#Open the test case list file that contains the list of
programs

open(TEST_LIST, "test_case_list_file");

#Open a merge file in write mode

$merged_tprof = "merge.tprof";

open(MERGE_FILE, ">$merged_tprof");

close(MERGE_FILE);

#Run each test case and collect coverage data

while(<TEST_LIST>)

  #configure coverage

  ATKConfigureCoverage($MyCCScripting, $CCS_DIR);

  #Enable Data Collection

  ATKStartDataCollection($MyCCScripting);
```

```
    #Run the program for the testcase

    …

    #Disable Data Collection

     ATKStopDataCollection($MyCCScripting);

    #Trace data(Tprof file) is available at this time in the

    #directory of the "program.out" with the name "pro-
gram.tprof"

     #set that as $curr_tprof

     #Merge the coverage data with the file

      ATKMergeCoverage($CCS_DIR, $curr_tprof, $merged_tprof);

     #Reset the CPU

}
#Process the merged trace data and convert it to excel
format

ATKTprof2xls($CCS_DIR, $MyProgram, $merged_trof,
$src_path);
```

Examples

C.1 Tree Example

Located in:

C64xx:
 <CSS– installation – directory>\examples\sim64xx\code_coverage\tree

C62xx:
<CSS– installation – directory>\examples\sim62xx\code_coverage\tree

C55xx:
<CCS–installation–directory>\examples\sim55xx\code_coverage\tree

This example is based on a binary search tree (BST). Inputs are read from a data file and inserted into the BST. After all the inputs are read from the file, the tree is traversed in in–order, pre–order and post–order and the results are printed on the standard output. The input data files should contain a list of integers separated by space or new line.

This example takes as input a file named "testcase_list" in the project directory. This file contains a list of input data file paths. Each file in the list is fed to the BST. A BST is created and traversed for every input data file.

See also Modified Tree Example below.

C.2 Interpreter Example

C64xx:

<CSS– installation – directory>\examples\sim64xx\code_coverage\ interpreter

C62xx:

<CSS– installation – directory>\examples\sim62xx\code_coverage\ interpreter

C55xx: <CCS–installation–directory>\examples\sim55xx\code_coverage\ interpreter

This example is a simple interpreter for a representative assemble language. The language supports the following instructions.

Instructions	Example Syntax	Meaning
MOV	MOV A,B;	A = B
MOVI	MOVI A,100;	A = 100
ADD	ADD A B;	A = A + B
ADDI	ADDI A 100;	A = A + 100
SUB	SUB A B;	A = A − B
SUBI	SUB A 100;	A = A − 100
MULT	MULT A B;	A = A ∗ B
MULTI	MULTI A 100;	A = A ∗ 100
DIV	DIV A B;	A = A / B
DIVI	DIV A 100;	A = A / 100
READ	READ A;	Read A from stdin
WRITE	WRITE A;	Write A to stdout

❑ Registers available are A, B, C ....,Y, Z

❑ Comments begin with a semicolon(;).

❑ A new instruction should start from a new line.

The interpreter opens an input file, identifies the instructions, and executes them. The program takes as input a file named testcase_list in the project directory. This file contains a list of input data file paths. Each file in this list is fed to the interpreter. Note that the files are specified relative to the location of the pro–gram file.

Follow steps as specified in the quick tour (section 2.3) to identify the lines of this application not covered by the set of inputs available with the example as specified in the file named testcase_list mentioned above. It is required to cover all the source lines of the interpreter by writing testcases in the above language.

Below is a sample assembly code :

```
MOVI A 100; A = 100
MOVI B 200; B = 200
ADD A B; A = 300
MULT A B; A = 60000
SUB A B; A = 59800
WRITE A;
Write B;
```

C.3 Tree Batch Mode Example

C64xx:
<CCS–installation–directory>\examples\sim64xx\code_coverage\
tree_batch_mode

C62xx:
<CCS– installation – directory>\examples\sim62xx\code_coverage\
tree_batch_mode

C55xx: <CCS–installation–directory>\examples\sim55xx\code_coverage\
tree_batch_mode

This example uses the tree example to demonstrate the cumulative coverage data collection for an application that is run multiple times over multiple test inputs. The example has a file called testcase_list that has all the test input filenames. The application, tree.out, is run for each of these test input file contents and at the end, the coverage data cumulated over all the runs is generated in an Excel sheet. These steps are automated in the CCStudio scripting–based perl script, Cumulator.pl, present in this example directory.

The script performs the following steps:

1) Open CCS for the particular simulator configuration.

2) Open the tree project.

3) Load the program tree.out.

4) Open the "testcase_list" file.

5) Read one line (one testcase input file) from the file into another file called current_test.

6) Configure code coverage and enable the data collection.

7) Run the application for the inputs in the filename present in current_test file.

8) When the application has finished running to completion, stop the data collection.

9) Merge the coverage data of the current run with the coverage data collected so far.

10) Generate xls sheet for the current run's coverage data.

11) Move the tree_summary.xls sheet generated in Step 10 to another file name.

12) Reset and Restart the target.

13) Repeat steps 5 – 12 until all the lines are processed from the testcase_list file.

14) Process the cumulated coverage data and generate the Excel sheet.

C.4 Modified Tree Example

The tree example discussed in section C.3 is modified as follows to suit the cumulative coverage data collection.

1.See main.c file. Now, the current_test file is opened instead of the testcase_list file. The program opens the file mentioned in current_test file anduses its inputs. Every time the program is run from the script, current_test file has the next file from testcase_list as its contents.

2.The testcase_list file has all possible input testcase files.

The script uses the following functions from the perl module, ATK_functions.pm that comes with the ATK installation.

Steps to run this example:

1) To run this example program, it is essential that your machine has the CCStudio scripting package installed for the target that you are using. When installed, CCStudio scripting will be found in <CCS− installation − directory>\bin\utilities\ccs_scripting directory.

2) Open a MSDOS command window. Run the batch file, CCS_Scripting_Perl.bat, present in the Code Composer Studio scripting directory mentioned above, to get the necessary settings for using the scripting package.

3) Change to this example's directory if you are not already there.

4) Open the script, Cumulator.pl, in a text editor. Modify $CCS_DIR to point to the installation directory of the CCS you are using currently. Save the script.

5) Set up CCS for the C6416 Device Functional Simulator Little Endian configuration if you are using the example for sim64xx, or the C55xx Cycle Accurate Simulator if you are using the example for sim55xx. Save the set up.

6) Run the script, Cumulator.pl, as perl Cumulator.pl. The script runs the tree program for inputs from all the files and generates the cumulated coverage information in"tree_summary.xls" file in the program (.\debug) directory. In addition, the individual run's processed data are available in the files tree_summary1.xls, tree_summary2.xls, tree_summary3.xls and tree_summary4.xls.

7) View the cumulated coverage data from the tree_summary.xls sheet using MS Excel.

8) A log file named atk.log is dumped in the same program directory into which the standard out and error messages from the ATK functions are dumped. Consult this file if xls file is not generated or for any messages from the merge and post processing steps.

Trace Files Generated

The final coverage and exclusive profile data is captured in an MS Excel file. This file is a rendering of the raw coverage and exclusive profile data in the comma separated variables (csv) format as described below for easy visualization, browsing and post processing. For every source file, a corresponding csv file is generated and apart from that, a summary csv file is generated. In case MS Excel is not available or if you want the information provided in another format, the csv files contain all of the information needed and can be directly read or processed.

The tprof file is post processed to generate the csv files with all the coverage and exclusive profile data. As is indicated, these files follow the standard csv format. The contents of these files are as listed in the next sections.

Table D−1 documents the trace files that are generated.

> **Note:**
>
> The raw data from which the coverage and exclusive profile data is gener-ated is in a .tprof extension file. You do not need to look at the contents of this file; you can actually only work with the post processed data in the csv files.

Table D−1. Trace Files and Data Contained in Them

Trace Files	Column	Description	Data Type
Summary csv	1	Function name	String
	2	Name of the source file. The source file could be absent, in which case this would be empty.	String
	3	Line number in the source file. If the source file or the line number is unknown this would be empty.	Positive, decimal integer
	4	Size of the function in bytes	Positive, decimal integer
	5	Function's start address	String
	6	Number of times the function is executed	Positive, decimal integer
	7	Number of lines of code in the function	Positive, decimal integer
	8	Number of lines executed in the function	Positive, decimal integer
	9	Percent coverage of the function	Positive, decimal integer
	10	Number of source lines in the function	Positive, decimal integer
	11 and up	Various event counts named by the column heading	Positive, decimal integer
Source details csv	1	Minimum line count	Positive, decimal integer
	2	Maximum line count	Positive, decimal integer
	3	Line number	Positive, decimal integer
	4	Line itself	String
	5 and up	Various event counts named by the column heading	Positive, decimal integer