User Guide

Line coverage

Department:	Aalborg Wireless Center
Creation Date:	18 December, 2002
Last Modified:	18 December, 2002 by Kenneth Skou Pedersen
ID and Version:	8434.522.03.001
Status:	Being Processed

0 Document Control

0.1 Document History

ID	Author	Date	Status
8434.522.03.001	KSP	18 December, 2002	Being Processed

0.2 References, Abbreviations, Terms

[TI 8010.801] 8010.801, References and Vocabulary, Texas Instruments

Table of Contents

1 Introduction

This purpose of this document is to provide a means of measuring to what extent the lines of a specific code file or Visual Studio project is being executed. Using profiling this can be performed. This is supported by Visual studio 6. The profiler is an analysis tool that you can use to examine the run-time behaviour of your programs. By using profiler information, you can determine which sections of your code are being executed or working efficiently. Producing information showing areas of code that are not being executed or that are taking a long time to execute does this. Basically the profiler supports two types of profiling, which again can be divided into subcategories. This document will not examine all of the supported types but mainly concentrate on the following:

- Function profiling
    - Function timing
    - Function coverage
- Line profiling
    - Line coverage

As example function and line coverage profiling is useful for determining which sections of your code are not being executed while Function timing is useful for determining which sections of your code is being executed efficiently. This document will briefly explain the basics of line and function coverage. For examples on how to perform these measurements on existing entities please see the user guide located at \umts\Tool_Documents\2_Highlevel\8434_521_test_coverage_user_guide.doc.

2 Profiling

The profiling supported in Visual Studio 6 includes several different methods for measuring and gathering information on the code. This spans from function profiling to line profiling. As seen in the introduction function profiling can be divided into two parts namely function timing and function coverage. The first gathers information on how much time each function has consumed which is very useful when optimizing the code. Furthermore it collects information on how many times each function has been called. The latter determines whether a function has been executed.

Line profiling can also be divided into two parts, namely line counting and line coverage. The first counts the times each line has been executed and due to the structure of this method it is very slow. The latter is similar to that of function coverage as it only determines whether a line of code has been executed. Therefore the profiling overhead for line coverage is lower than for line counting, because the profiler needs to stop at a line only once.

Even though a design does not receive 100 % coverage it is still not only a powerful tool for determining holes in the test suite it is also a method of documenting the level of test from time to time. That is when new test cases are added it is possible to measure and document whether ones new test cases actually tests untested code or already tested code.

2.1 Line coverage

Line coverage is useful for determining which sections of your code are not being executed by default. The profiler lists all profiled lines, with an asterisk (*) marking those that were executed. If a line is not executed during tests, the code has not been fully tested. This way line coverage is useful for determining holes in the test suite.

To illustrate this a small example of a sample line coverage report is given here:

```
1: * a=0;
2:
3: * if (a)
4: {
5: b=2;
6: }
7: * c=b;
```

In the previous all lines except number 5 have been executed. As can be seen this is easy to cope with for small code samples. However when it comes to large code files, which will be the case this way of presenting the results will fail. In addition to this many people will probably have difficulties in distinguishing between the line that has not been executed and the lines with no code generated. An example of such a line could for instance be line 4 or 6. A solution to this will be presented further down in section 2.3.

2.2 Function coverage and function timing

Function coverage is useful for determining which functions of your code that are not being executed. Basically it works like line coverage described above so it will not be explained in further detail here but merely described how to perform the measurements.

While function and line coverage are useful for determining holes in the test suite function timing on the other hand is useful for optimizing the code. Performing this method results in a list of all functions and how much time each function has consumed starting with the most time consuming function. In addition to this it also states how many time the individual function has been called.

2.3 Presenting the results

The default method of representing the results might seem fine at first sight but when the coverage includes a lot of code files the results can be very comprehensive and hence difficult to cope with. Another solution would be to filter out the lines that have been executed. In addition to this it would be smart to arrange the remaining line so that it would be possible to access them, simply by double clicking on them much in the same way it works for debugging in Visual Studio.

A fairly simple PERL script could perform this task by processing the final results from the coverage. The output of this script should be a file format that could be used in Visual Studio. By loading this file into Visual Studio it would then be possible to quickly jump to the lines in question.

3 How to perform the profiling

There are 3 phases to the profiling process, namely pre-runtime, runtime and post-runtime. These phases has to be wrapped around the stack test process in order to perform the actual profiling. Using a pre-created batch job, which will automate most of the process, is the easiest way to perform this task. The details of this batch job will only be examined briefly further down in this document in section 3.7.

When deciding upon how to perform the profiling several issues arise. One of the main issues is how to start the actual profiling and how much of this process can be automated. When it comes to start the profiling this can be done from Visual studio 6, command prompt or the TAPcaller. However this document will only describe the use of the TAPcaller, as this is the most suitable. Another important issue is how to execute the test cases regarding the protocol stack.

Basically there are 2 different approaches to execute the test cases. Either the protocol stack is restarted for each test case or all the test cases are executed without restarting the stack. Within an ideal code-world there would not be any difference. Naturally this is not the case and some of the entities or test cases in the UMTS project are highly dependent on whether the stack is restarted for each test case.

Naturally the goal is to eliminate this problem but as this requires a great deal of work for some entities this is probably not within the near future. Therefore it is necessary to take both approaches into consideration. These two methods will briefly be examined in section 3.2 and 3.3. In addition to this there are some steps and requirements, which are common for the 2 approaches. These steps will briefly be examined first.

3.1 Common requirements for the 2 approaches

No matter which approach for performing the profiling that is chosen there are some simple steps, which has to be performed by the developer. The first step is to enable profiling in Visual Studio 6. This is done by selecting the project properties: “Project” | “Settings” | ”link tab”. Make sure that “Enable profiling” and “Generate .map file” both are checked. This is illustrated in Figure 1. Please note that it is necessary to rebuild the stack after changing these options for the profiling to work. Note that these settings can be leaved in for normal use as well.

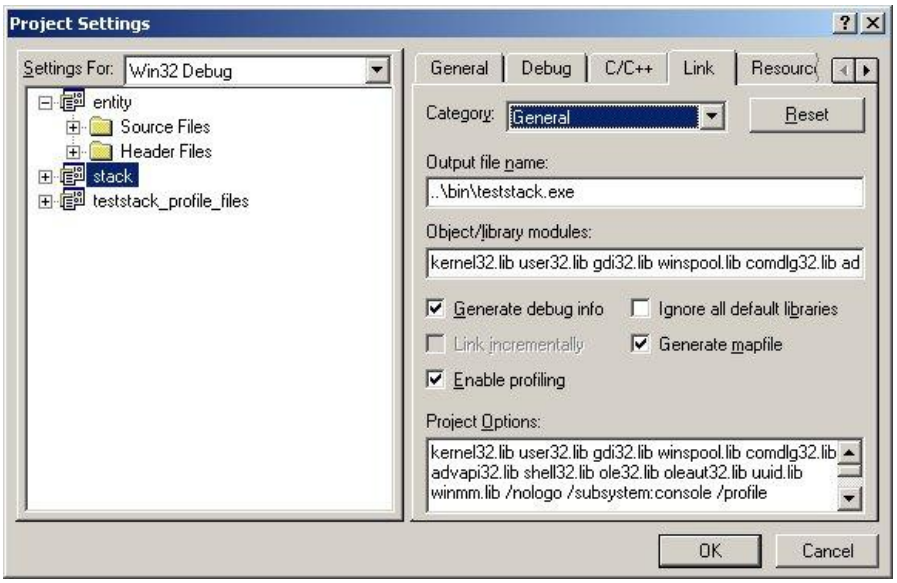

Figure 1: Enabling profiling in Visual Studio 6.

The second job for the developer is to specify which source files are to be included in the profiling. This is necessary as it not feasible to profile on all files each time, as this will simply make the result very comprehensive and hence time consuming and confusing. However it would probably be manageable to

include an entire entity. Naturally the developer also has to specify which protocol stack executable he/she wants to profile. The simplest way to specify this would be in a simple text file instead of arguments to executable. This way the information can easily be modified and reused. The format for defining the different parameters will be described further down in section 3.5.

3.2 Approach 1: Restarting the protocol stack each time

The first approach to be examined is with a restart of the protocol stack for each test case. In order to automate and rationalize the profiling process the TAPcaller will be used to execute the test cases. This is also convenient as this is currently supported by the existing version of the TAPcaller.

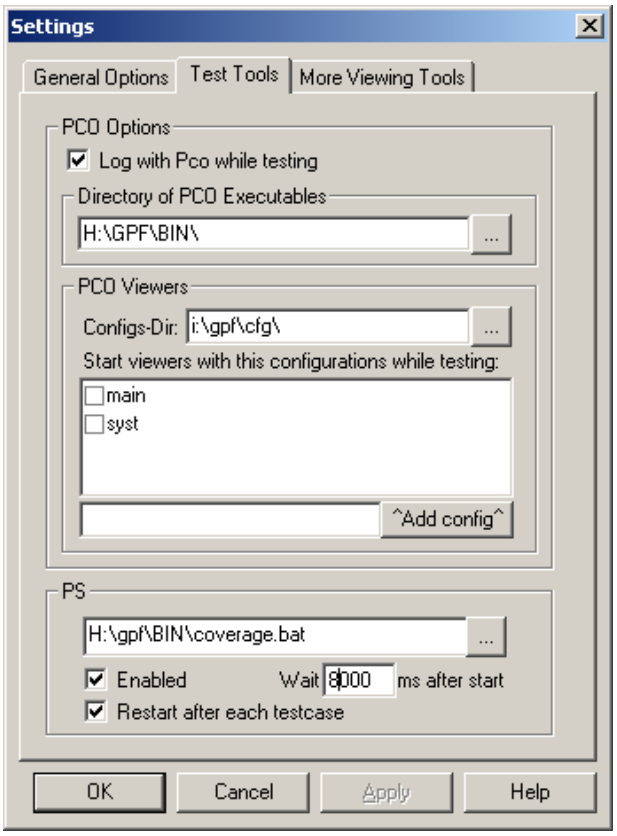

Figure 2: Configuring the TAPcaller.

In order to make the TAPcaller start the protocol stack for each test case it is necessary to change the default configuration. This is done by selecting the “Configuration” | “Settings” | “Test Tools” tab. In the lower part it is possible to specify which protocol stack is to be started for each test case. The “Enabled” option has to be checked in order to activate this option. However instead of specifying the actual protocol stack one can specify the profiling batch job, which will start the profiling.

In addition to this it is probably necessary to increase the delay defined in the lower right part in order to allow for the profiling to start before executing the test case (In this case it has been increased to 5000 ms, This may vary depending on the computer speed and hence it might be necessary to adjust the delay). This way a profiling will be carried out for each test case. However as a profile per test case is not very useful the different profiling result will continuously be merged in order to generate a complete result for all the test cases executed.

3.3 Approach 2: The protocol stack is started only once

In this approach the protocol stack is only started once and then all test cases are executed. The TAPcaller also supports this scenario when specifying which protocol stack to use as it is possible to check/uncheck whether to restarts the stack for each test case. Following the feature used to start the profiling job as described in the previous section can also be used but the profiling batch job can also be started manually before executing all of the test cases. After executing the test cases the protocol stack should manually be closed in order to finish the profiling. This way the entire profiling of the source can be done without merging the results continuously and hence save some time.

3.4 Future support of profiling

Another important issue is how this profiling relates to Visual studio 7. Will it be possible to use the same tools or is it necessary to use some 3rd party tool? Both may result in changes to the batch job. According to the documentation for Visual studio 7 it should be possible to use but at least for time profiling better free tools exist. (TBD)

3.5 How to define the source files to be profiled

In order to perform the profiling it is necessary to specify the desired code and the executable to be profiled. This information could be passed on as arguments when executing the profiling job. However this is not very feasible, as it would result in a very long and complicated argument list. Instead specifying the source code to be measured in a specific list file does this. Such a list file should exist for each entity so it can be used from time to time. In addition to this there can be several different list files for each entity for different measurements.

The list files should be saved in a simple text file with suffix .lst such as for instance sm.lst. This way it will be fairly simple to edit the arguments as it can be done in any text editor and it will be possible to reuse the information and hence it will be easy to run several profiling measurement under the same conditions. Following the result can be compared directly.

The source files should be included one by one with one source file per line. This will e.g. result in a file like the following:

sm_cof.c(0-0)
sm_cop.c(0-0)
sm_cos.c(0-0)
sm_kef.c(0-0)

The (0-0) indicates that all lines in the file will be profiled. Other combinations such as (10-20) can also be used and would result in that line 10 to 20 will be profiled.

The created list files should be placed together with the source codes for the test cases. That is for the SM entity the list files should be placed under \g23m\Condat\ms\src\sm\test_usm\. This is also where the result files will be placed.

3.6 The actual profiling

After specifying the different detail above you are now ready to perform the profiling. The first thing to do is to determine whether the stack should be restarted each time and what kind of profiling is desired. The profiling only differs slightly depending on which method is chosen. After specifying the details in the profile_config.bat file the profiling generally consists of 3 steps:

1. Start the setup of the chosen method e.g. setup_line_coverage.bat. This should be done from a 4nt prompt in order to ease the error handling if any necessary. Initumts.bat should be altered so that it is possible to execute the batch files from any directory.

2. If the stack is only to be started once setup the TAPcaller accordingly or run the coverage.bat file manually and execute the desired test cases. This is described in section 3.2. After setting up the TAPcaller or starting coverage.bat manually execute the desired test cases.
3. Finish the profiling process by executing the post processing batch file post_processing.bat.

The profiling has now finished and the results are saved in a file called exe-name-LV.lst, exe-name-LV2.lst and in a file called exe-name-LV.out. The first file contains the statistics such as lines of code and percentage run while the second only is used for generating output for the Visual Studio debugger (This last part is currently only applicable for the line coverage method).

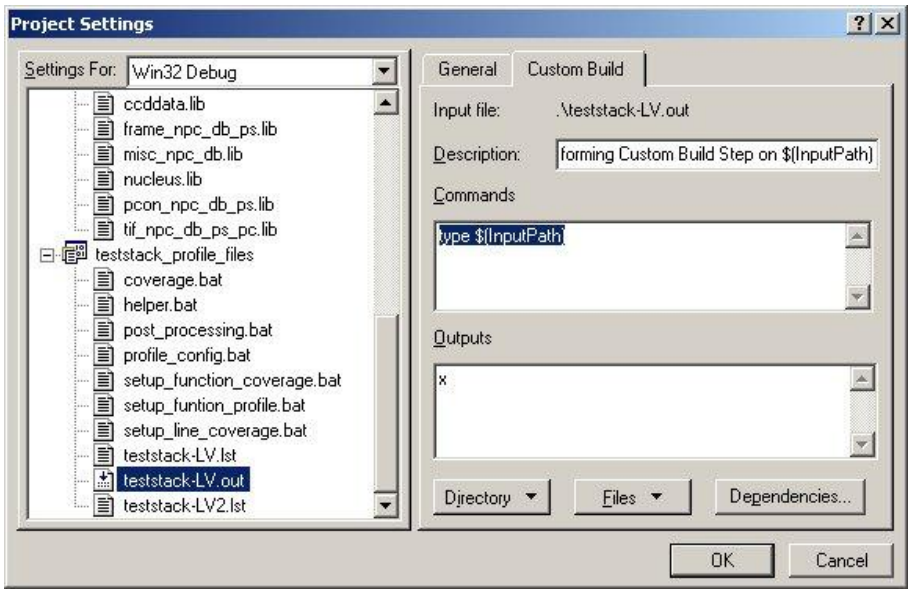

Figure 3: Setting the properties for the output file.

The last file can be added to Visual studio projects for easier access to the line in question. By adding the file to a Visual Studio project and setting the properties as illustrated in Figure 3. This way it is possible to send the output file to the debugger window by compiling this file.

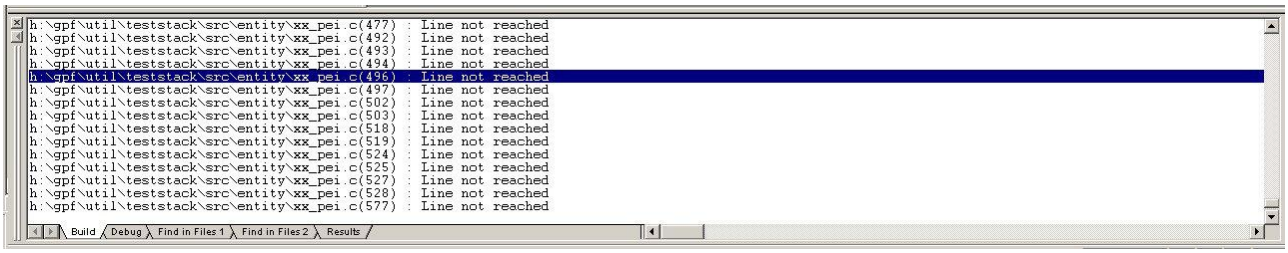

Figure 4: Illustration of how the result when compiling the output file.

Following it is possible to double click on the lines for quick access to the line in question. This is illustrated in Figure 4. Currently this is only available for line coverage.

3.7 Pseudo code for the profiling batch jobs

This section will briefly describe the code/steps needed in order to perform the profiling in pseudo code. The details in the following are not intended for the developer but merely to clarify the structure for future updates performed by the tool group. This structure is common for all the profiling methods described above. Naturally there will be small differences in the arguments such as for instance the method used. However this is fairly simple and will not be described any further.

3.7.1 The setup batch file:

Get variables from profile_config.bat file

Get source files to profile from the .lst file.

Copy the .exe and .map file

PREP normally

3.7.2 The coverage batch file:

Close any open instances of %exe%.exe

If not first run then PREP normally

Start helper.bat file

3.7.3 The helper batch file:

Profile %exe%

Exit

3.7.4 The post-processing batch file:

Close any open instances of %exe%.exe

PREP normally

PLIST %exe%

Process output with perl script.

Clean up in old files

Delete %exe%-merged

4 Appendix

This appendix shortly lists the used parameters in PREP, PROFILE and PLIST command respectively. This section is intended for reference only and is part of the tables in the Visual Studio help. If you want more information on PREP, PROFILE and PLIST consult the MS Visual C++ help.

4.1 PREP

PREP [options] [programname1] [programname2...programname8]

Option	Phase		Description
	I	II	
/AT	X		Collects attribution data for function timing and function counting. Function attribution reports which function called another function. See the /STACK switch later in this list.
/CB	X		Used with function timing, allows you to set the calibrated overhead of profiler calls in the event that your function timing calls have varied because of varied calibrated overhead values. The calibrated overhead is displayed in default (non-tab-delimited) PLIST output
/EXC	X		Excludes a specified module from the profile. You can also exclude part of a module. For example, /EXC test.cpp(9-18).
/EXCALL	x		Excludes all modules from the profile. Used with /INC.
/FC	x		Selects function count profiling.
/FT	x		Selects function timing profiling. This option causes the profiler to generate count information as well. This is the default setting from the command line.
/FV	x		Selects function coverage profiling.
/INC	x		Includes in profile. Used in conjunction with /EXCALL.
/H[ELP]	x	x	Provides a short summary of PREP options.
/IO *filename*		x	Merges an existing .PBO file (PROFILE output). Up to eight .PBO files can be merged at a time. The default extension is .PBO.
/IT *filename*		x	Merges an existing .PBT file (PREP Phase I output). Up to eight .PBT files can be merged at a time. You cannot merge .PBT files from different profiling methods. The default extension is .PBT.
/LC	x		Selects line count profiling. May take a long time.
/LV	x		Selects line coverage profiling.
/M *filename*		x	Substitutes for the /IT, /IO, and /OT options.
/NOLOGO	x	x	Suppresses the PREP copyright message.
/OI *filename*	x		Creates a .PBI file. The default extension is .PBI. If /OI is not specified, the output .PBI file is *programname1*.PBI.
/OM	x		Creates a self-profiling file with an _XE or _LL extension for function timing, function counting, and function coverage. Without this option, the

			executable code is stored in the .PBI file. This option speeds up profiling.
/OT *filename*	x	x	Specifies the output .PBT file. The default extension is .PBT. If /OT is not specified, the output .PBT file is *programname1*.PBT.
/SF *function*	x		Starts profiling with *function*. The *function* name must correspond to an entry in the .MAP file.
/STACK *dpt*	x		When using the /AT switch, you can also set the stack depth (*dpt*) to which functions will have their attribution data recorded.
/?	x	x	Provides a short summary of PREP options.

4.2 PROFILE

PROFILE [options] programname [programargs]

Options	Description
/A	Appends any redirected error messages to an existing file. If the /E command-line option is used without the /A option, the file is overwritten. This option is valid only with the /E option.
/E *filename*	Sends profiler error messages to *filename*.
/H[ELP]	Provides a short summary of PROFILE options.
/I *filename*	Specifies a .PBI file to be read. This file is created by PREP.
/NOLOGO	Suppresses the PROFILE copyright message.
/O *filename*	Specifies a .PBO file to be created. Use the PREP utility to merge with other .PBO files, or to create a .PBT file for use with PLIST.
/X	Returns the exit code of the program being profiled.

4.3 PLIST

PLIST [options] inputfile

Options	Description
/C *count*	Specifies the minimum hit count to appear in the listing.
/D *directory*	Specifies an additional directory for PLIST to search for source files. Use multiple /D command-line options to specify multiple directories. Use this option when PLIST cannot find a source file.
/F	Lists full paths in a tab-delimited file.
/FLAT	When using function attribution (see PREP /AT), displays function attribution with no indentation.
/H[ELP]	Provides a short summary of PLIST options.
/INDENT	When using function attribution (see PREP /AT), displays function attribution information in indented format. This is the default display for function attribution if neither /FLAT nor

	/TAB is selected.
/NOLOGO	Suppresses the PLIST copyright message when output is directed to the screen.
/PL length	Sets page length (in lines) of output. The length must be 0 or in the range 15 through 255. A length of 0 suppresses page breaks. The default length is 0.
/PW width	Sets page width (in characters) of output. The width must be in the range 1 through 511. The default width is 511.
/SC	Sorts output by counts, highest first
/SL	Sorts output in the order that the lines appear in the file. This is the default setting. This option is available only for line profiling
/SLS	Forces line count profile output to be printed in coverage format
/SN	Sorts output in alphabetical order by function name. This option is available only for function profiling.
/SNS	Displays function timing or function counting information in function coverage format. Sorts output in alphabetical order by function name.
/ST	Sorts output by time, highest first.
/T	Tab-separated output. Creates a tab-delimited database from the .PBT file for export to other applications. All other options, including sort specifications, are ignored when using this option.
/TAB *indent*	When using function attribution (see PREP /AT), sets tab width for indentation of function information.
/?	Provides a summary of PLIST options.