Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ Code Coverage Profiler

User Guide

80-N2040-20 Rev. E

September 24, 2018

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

1 Introduction

This document describes the Qualcomm® Hexagon™ code coverage profiler, which displays information on the execution history of a program written for the Hexagon processor.

This document is a reference for C programmers who have assembly language experience.

1.1 Conventions

`Courier` font is used for computer text and code samples:

```
unsigned long long hexagon_sim_read_pcycles()
```

The following notation is used to define command syntax:

- Square brackets enclose optional items (e.g., `[label]`).
- **Bold** indicates literal symbols (e.g., `[`**`comment`**`]`).
- The vertical bar character, `|`, indicates a choice of items.
- Parentheses enclose a choice of items (e.g., `(add|del)`).
- An ellipsis, `. . .`, follows items that can appear more than once.

1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Overview

For a program written for the Hexagon processor, the code coverage profiler can display the following information:

- Cycle counts for individual instructions
- Cycle counts for functions
- Program-relative cycle percentages for functions

The profiler operates as a standalone utility program that can generate the profiling information in HTML or plain text format. It can profile all Hexagon processor applications, whether standalone, RTOS-based, or single- or multi-threaded.

NOTE: The profiler performs *postmortem* processing of the target application: that is, it is used after the target application has completed executing.

2.1 Profiling tool

The primary profiling tool for the Hexagon processor is the gprof profiler. For more information, see the *Hexagon gprof Profiler User Guide*.

2.2 Profile data files

The code coverage profiler obtains its profiling information from data files (named `gmon`) that are generated by the Hexagon simulator. For more information on generating profile data files, see the *Hexagon Simulator User Guide*.

2.3 Profile reports

The code coverage profiler generates profiling information output in a report. Following are the report formats:

- Plain text – a single text file
- CSV – a plain text file with comma-separated values (CSV format)
- HTML – of a top-level index file and a directory of the remaining HTML files

For details, see Section 3.5.

3 Using the Profiler

Profiling a program is a two-step procedure:

1. Execute the program on the simulator with profiling enabled to generate a profile data file (Section 3.1).
2. Run the code coverage profiler to analyze the profile data (Section 3.2).

The profiler generates two types of profile information:

- *Annotated disassembly* listing, in which the source code is intermixed with the disassembly information. For details, see Section 3.5.2.1.
- *Function statistics* that display information about the functions in an application. For details, see Section 3.5.2.2.

3.1 Create the profile data file

Before an application can be profiled, a profile data file must first be created for it.

To create a profile data file, simulate the application with the appropriate command option to generate the profile data file. The simulator generates profile data files in gmon format (which are typically used by the gprof profiler).

The code coverage profiler works with gmon files.

3.1.1 Create gmon files

When the simulator is invoked with the `--profile` command option, it generates one or more gmon profile data files while it executes the target application. In some cases, a program's profile data is distributed across multiple profile data files:

- When a program is a multithreaded standalone program, its profile data files are named for each hardware thread (such as `gmon.t_0`, `gmon.t_1`, ...).
- When a program is a multithreaded RTOS application, its profile data files are named for each software thread (such as `gmon.name1`, `gmon.name2`, ...).

In these cases, all of the profile data files must be specified on the profiler command line. For simplicity, the relevant command option accepts wildcard characters (e.g., `gmon_t*`).

For more information on profile data files, see the *Hexagon Simulator User Guide*.

3.2 Start the profiler

To start the code coverage profiler from a command line, type:

```
hexagon-coverage (-i executable_file | -d disassembly_file)
            [-o outfile | -c csv_outfile | --html html_outpath]
            gmon_data_files ...
            [options]
```

The program to be profiled must be specified with one of the following files:

- An executable file (using `-i`)
- A disassembly listing file (using `-d`)

Specify the profile report with one of the following formats:

- Plain text report (using `-o`)
- CSV report (using `-c`)
- HTML report (using `-html`)

The profile data files specified must be gmon files. All of the profile data files for the application must be specified. Use wildcards to specify multiple profile data files (e.g., `gmon_t*)`.

The remaining options are used to control the profiler output or the contents of the profile report. (One exception, `--mapfile`, is used when the code coverage profiler is run on a different machine than the machine the application was built on.)

If no profile report option is specified, a plain text report is written to the standard output by default. If more than one profile report option is specified, only the last one on the command line is recognized.

NOTE: The profiler must be run in a directory where you are able to create files.

NOTE: CSV reports contain function statistics only; they do not include program code.

3.3 Profiler options

The code coverage profiler options are used to control various profiling features from the command line.

The options are specified by the command switches listed below. They can be specified in any order.

-c *filename* | **--csv-file** *filename*
--color-max *value*
--color-min *value*
-d *filename* | **--disasm_file** *filename*
-D | **--no-disasm**
-F | **--no-funcs**
-h | **--help**
--html *[pathname]*
-i *filename* | **--image** *filename* | **--symfile** *filename*
-m *filename* | **--mapfile** *filename*
-o *filename* | **--outfile** *filename*
-q | **--quiet**
-S | **--no-source**
-v | **--version**
--verbose

Details:

-c *filename*
--csv-file *filename*

Write the profile report to the specified file in comma-separated values (CSV) format.

CSV reports contain function statistics only.

--color-max *value*

Specify the threshold value for the maximum code coverage level (green) in an HTML profile report. The default value is 80.

Numeric code coverage levels are color coded in HTML reports using green, yellow, and red. Green indicates the highest coverage levels, red indicates the lowest levels, and yellow indicates all levels in between.

--color-min *value*

Specify the threshold value for the minimum code coverage level (red) in an HTML profile report. The default value is 20.

Numeric code coverage levels are color coded in HTML reports using green, yellow, and red. Green indicates the highest coverage levels, red indicates the lowest levels, and yellow indicates all levels in between.

-d *filename*
--disasm_file *filename*

Specify the input disassembly listing file.

Disassembly listings are created by the object file viewer utility. For more information see the *Hexagon Utilities User Guide*.

This option is not used if `-i` is used.

-D
--no-disasm
Suppress the annotated disassembly listing in the profile report.

-F
--no-funcs
Suppress function summaries in the profile report (plain text format only).

-h
--help
Display code coverage profiler command options, and exit.

--html *[pathname]*
Write the profile report to the specified path in HTML format (Section 3.5). The default path is the current directory.

-i *filename*
--image *filename*
--symfile *filename*
Specify the input executable file.

This option is not used if `--disasm_file` is used.

-m *filename*
--mapfile *filename*
Specify the map file used to locate the application source code when an application is profiled on a different machine than the machine it was built on (Section 3.4).

-o *filename*
--outfile *filename*
Write the profile report to the specified file in plain text format.

When no report option is specified, the profile report is written to the standard output in plain text format.

-q
--quiet
Suppress the overall summary statistics line in the profile report (plain text format only).

-S
--no-source
Suppress the annotated source code listing in the profile report.

-v
--version
Display the version number of the code coverage profiler, and exit.

--verbose
Display the name of each file and function as it is processed.

3.4 Input files

Profile data files

The code coverage profiler accepts as input one or more profile data files in gmon format (Section 3.2).

Disassembly listing file

If the code coverage profiler is invoked with the `-d` option (Section 3.3), it requires a disassembly listing file of the program being profiled. The profiler uses the symbols defined in this file to generate its profiling information.

Disassembly listings are created by the object file viewer utility. For more information, see the *Hexagon Utilities User Guide*.

Map files

If an application is built on one machine and then profiled on another machine, the pathnames to the source code files that are stored in the executable's debug information might no longer be valid on the profiling machine. In this case, the profiler will include only the disassembled code in a profile report.

If this problem occurs, a map file can be created. This file maps the source code directories on the build machine (the application's build pathnames) to the corresponding directories on the profiling machine (the application's profile pathnames).

This map file is a plain text file. Each line in the file contains the root path on the build machine followed by the root path on the profile machine, with the two paths separated by an `=` character.

For example:

```
/qc/hexagon/user = C:\Temp\src\dsp\prj\user
/dsp/core/kernel/qurt/ = C:\Temp\src\qurt\
```

When this file is specified on the command line, the profiler can locate the application source code and include it in the profile report.

3.5 Output files

The profiling information generated by the code coverage profiler is output in a report that presents the program source code and disassembled object code, along with metadata appended to each line indicating the cycle counts and program-relative cycle percentages for the associated code.

3.5.1 Profile report formats

Profile reports can be generated in the following formats.

3.5.1.1 Plain text reports

Plain text reports consist of a single text file (which defaults to `stdout`). These reports contain the annotated source and disassembly code for the application, followed by a summary detailing the coverage for each source file or function.

3.5.1.2 CSV reports

CSV reports consist of a plain text file in comma-separated values (CSV) format. These reports contain function statistics only, and do not include any program code.

3.5.1.3 HTML reports

HTML reports consist of a top-level index file and a directory containing the remaining HTML files. A separate HTML file is created for each source file. If no source file information is available for a given function, a separate HTML file is generated for the function.

Each HTML file contains the annotated source and assembly code for the corresponding source file, followed by a summary detailing the coverage for each source file or function.

When the top-level index file is opened in a web browser, the HTML report displays two panes:

- A table of contents (TOC) pane listing links to all the source files and functions in the application. The TOC is paginated, and can be both searched and sorted.
- A source pane displaying the annotated source and disassembly code for the application. The disassembly code can be hidden so only the source code is visible.

In both the TOC pane and source pane, user-definable color coding (Section 3.3) indicates the level of coverage in each code line.

Figure 3-1 through Figure 3-3 show example pages in an HTML report.

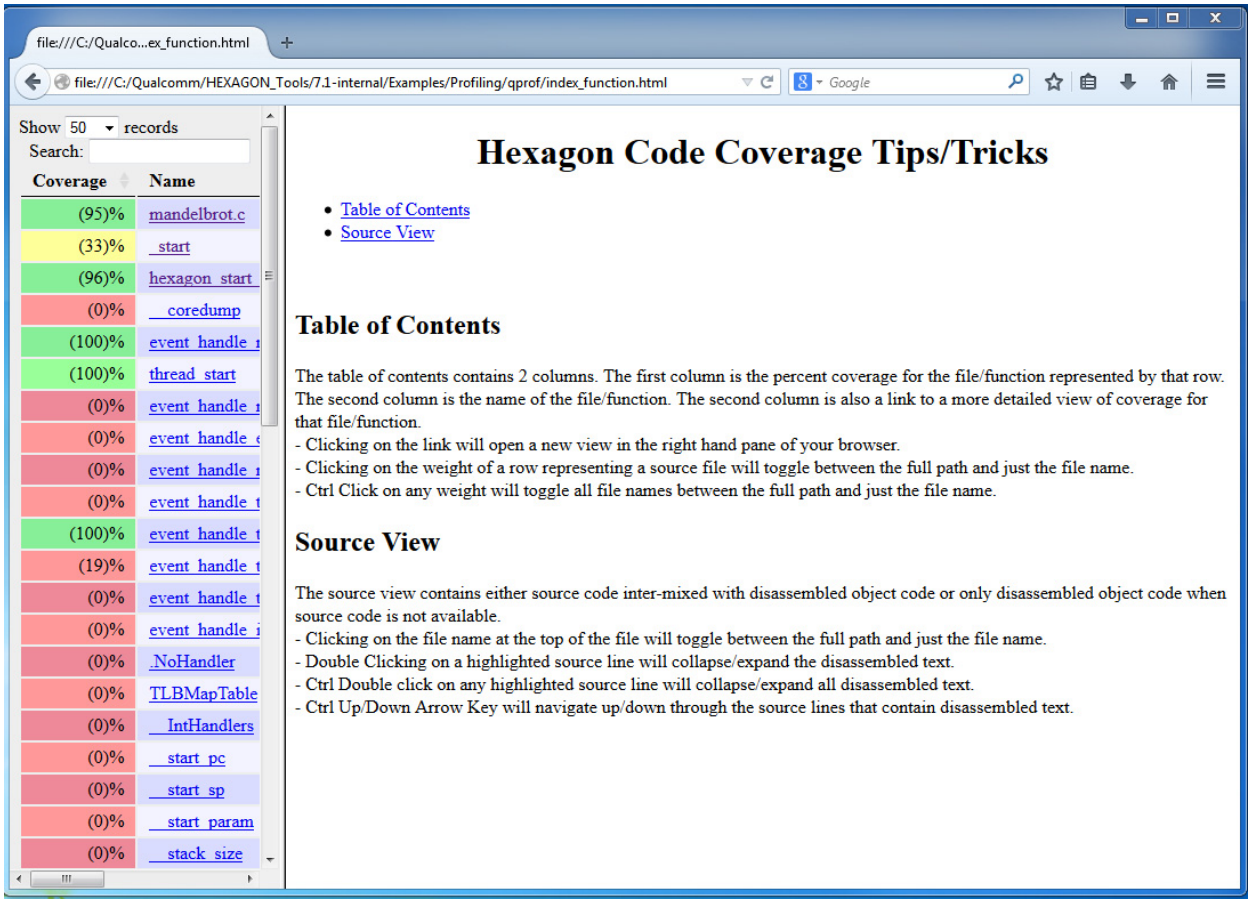

Figure 3-1 HTML report (initial view)

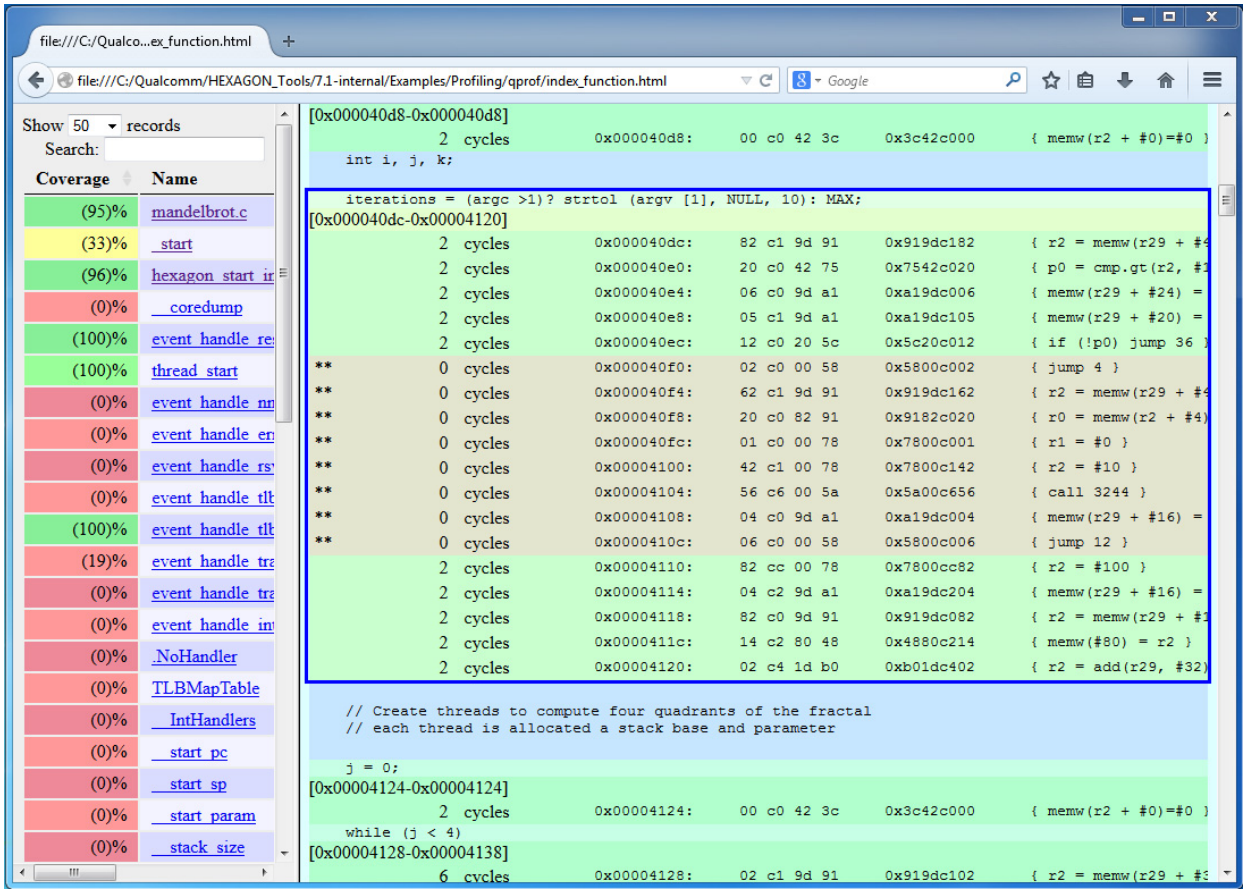

Figure 3-2 HTML report (annotated code)

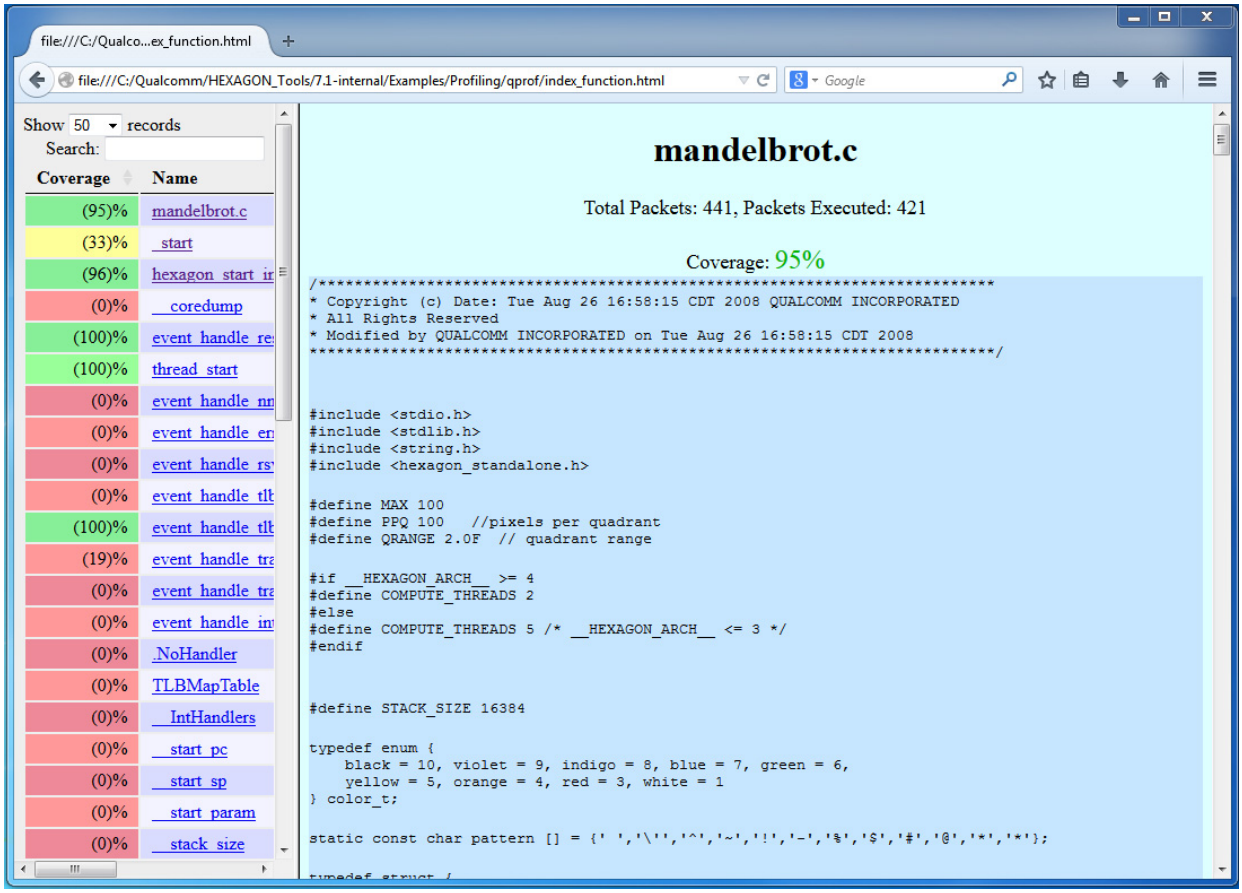

Figure 3-3 HTML report (file summary)

3.5.2 Generated profiling information

3.5.2.1 Annotated disassembly

In an annotated disassembly listing, the source code is intermixed with the disassembly information (provided that the source code is accessible and was produced with the `-g` option).

Each Hexagon processor instruction packet is prefixed with the cycle count for that packet.

Packets that never executed are marked at the beginning of the line with `**`.

```
.
.
.
00005000 <main>:
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
        20 cycles     5000: 10 41 00 5a     5a004110     { call 5220 <puts>
                      5004: 60 45 80 49     49804560       r0 = memw (gp + #172)
                      5008: 00 c0 9d a0     a09dc000       allocframe (#0) }

    return 21;
}
         2 cycles     500c: a0 42 00 78     780042a0     { r0 = #21
                      5010: 1e c0 1e 90     901ec01e       deallocframe }
         1 cycles     5014: 00 c0 9f 52     529fc000     jumpr r31
**       0 cycles     5018: 00 40 00 7f     7f004000     { nop
                      501c: 00 c0 00 7f     7f00c000       nop }

00005020 <thread_create>:
**       0 cycles     5020: 00 40 24 72     72244000     { r4.h = #0
                      5024: 47 42 02 8c     8c024247       r7 = asl (r2, #2)
                      5028: 00 40 25 72     72254000       r5.h = #0
                      502c: 00 c0 26 72     7226c000       r6.h = #0 }
.
```

3.5.2.2 Function statistics

Following the disassembly list (Section 3.5.2.1) in the generated profiling information is the list of functions with their statistics:

- The first line displays the function name, its address range, and any aliases it is known by.
- The second line shows the total number of packets within the function's address range, the number of these packets that were executed (cycle count > 0), and the packet coverage.

Packet coverage is calculated as (packets executed) / (total packets), and is expressed as a percentage value.

```
Function '_start' (0x00000000-0x00000018) is aliased with 'start':
    TotalPackets =       3, packets executed =       1, coverage =  33.33%
Function 'hexagon_start_init' (0x00000098-0x0000056c):
    TotalPackets =     310, packets executed =     301, coverage =  97.10%
Function 'coredump' (0x00000570-0x00000760) is aliased with '__coredump':
    TotalPackets =     125, packets executed =       0, coverage =   0.00%
<snip>
Function 'sys_Tlsget' (0x0000d590-0x0000d5ec):
    TotalPackets =      13, packets executed =       0, coverage =   0.00%
Function 'sys_write' (0x0000d5f0-0x0000d664):
    TotalPackets =      19, packets executed =      14, coverage =  73.68%
Function '_fini' (0x0000d680-0x0000d6b4):
    TotalPackets =      12, packets executed =      11, coverage =  91.67%
Function 'Unaccounted for packets' (0xffffffff-0xffffffff):
    TotalPackets =     792, packets executed =       0, coverage =   0.00%
```

The special function name, `Unaccounted for packets`, displays the number of packets encountered in the program that were not within any defined function address range. This case can be due to data existing within the `.text` section, or to some other reason (such as TLB entries in the case of the `crt0_standalone .S` file).

The final line in the function statistics is an overall summary of the function statistics. It has the following form:

```
*-*- Summary -*-*  Total packets = 6265, packets executed = 1005,
coverage =  16.04%
```

This line displays the following information:

- Total number of packets encountered
- Number of packets actually executed
- Coverage percentage: `(packets_executed) / (total_packets)`

NOTE: The function statistics can be written to a separate file (in `csv` format) by using the `-c` option (Section 3.3).