Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ Resource Analyzer

User Guide

80-N2040-21 Rev. D

December 2, 2019

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

Contents

1 Introduction

This document describes the Qualcomm® Hexagon™ processor Resource Analyzer, a software tool that enables you to analyze memory resources being used in a Hexagon program.

1.1 Conventions

- Courier font is used for computer text, for example, `printf("Hello world\n");`.
- The following notation is used to define the syntax of functions and commands:
  - Square brackets enclose optional items, for example, **`help`** `[command]`.
  - **`Bold`** is used to indicate literal symbols, for example, the brackets in *`array`***`[`***`index`***`]`**.
  - The vertical bar character, `|`, is used to indicate a choice of items.
  - Parentheses are used to enclose a choice of items, for example, `(`**`on`**`|`**`off`**`)`.
  - An ellipsis, `. . .`, follows items that can appear more than once.
  - *`Italics`* are used for terms that represent categories of symbols.

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

2.1 Features

The Resource Analyzer supports the following features:

- Analysis of either a single ELF file or multiple program files stored in a source code directory structure
- Listing of program functions (sorted by name and by size)
- Listing of function call paths (sorted by cumulative stack use)
- Listing of unknown functions

2.2 Limitations

The Resource Analyzer is intended for use in calculating the worst-case stack requirements of a program's software threads. However, the following limitations make this task more difficult:

- The mechanisms for thread creation and stack allocation are specific to the operating system (OS) that is used with the program
- The wrappers used by the program to access the OS are also OS-specific

The Resource Analyzer cannot identify which code trees have separate stacks allocated for them and which do not. In practice, this is not a problem because software threads typically appear in the Resource Analyzer as both root nodes and the largest consumers of stack memory. Thus, using the Resource Analyzer to list a program's functions sorted by cumulative stack use will cause the software threads to appear at the top of the list.

3 Using the analyzer

After building a program, use the Resource Analyzer to analyze that program and display its resource usage.

The analyzer can input programs in either of the following forms:

- A single binary ELF file
- Multiple files stored in a program's source code folder

The analyzer operates as a standalone command line tool. The resource usage information is generated in several text files, with the file data in an easy-to-read format.

3.1 Start command

To start the Resource Analyzer from a command line, enter the following command:

```
hexagon-analyzer-backend [option...]
```

Use command switches to specify the input and output files for the program, and any additional command options (Section 3.2). A switch consists of one or two dash characters followed by a switch name and parameter. Switch names are case sensitive, and the switches must be separated by at least one space. For example:

```
hexagon-analyzer-backend --dsp v68 --folder my_prog
```

To list the proper command line syntax and available command options, use the `--help` option:

```
hexagon-analyzer-backend --help
```

3.2 Options

The options are specified by the command switches listed below. Some options have alternate abbreviated switches defined for ease of use.

Display information (Section 3.2.1)

```
--help | -h
--version | -v
```

Input files (Section 3.2.2)

```
--aidfile filename
--elffile filename
--folder pathname | -f pathname
--rootfile filename
```

Output files (Section 3.2.3)

```
--outdir pathname | -o pathname
```

Processor information (Section 3.2.4)

```
--dsp version
```

3.2.1 Display information

NOTE: These options are used as sole command arguments. They do not launch the Resource Analyzer.

--help
-h

Display the proper command line syntax and available options, and then exit.

--version
-v

Display the release version of the Resource Analyzer and exit.

3.2.2 Input files

--aidfile *filename*

Specify the file name of the aiding file. For more information on aiding files, see Section 3.6.

--elffile *filename*

Specify the file name of the program file to be analyzed.

--folder *pathname*
-f *pathname*

Specify the path name of the program folder to be analyzed.

--rootfile *filename*

Specify the file name of the root specification file. For more information on specification files see Section 3.7.

3.2.3 Output files

--outdir *pathname*
-o *pathname*

Specify the target directory for output files. For more information on output files see Section 3.5.

3.2.4 Processor information

--dsp *version*

Specify the Hexagon processor version of the program file to be analyzed: `v4`, `v5`, `v55`, `v60`, `v61`, `v62`, `v65`, `v66`, `v67`, `v67t` (for Small Core), `v68`.

For more information on the Hexagon processor versions, see the appropriate *Qualcomm Hexagon V6x Programmer's Reference Manual*.

NOTE: Not all of the `-dsp` option settings are supported in a specific Hexagon tools release. For more information, see the *Qualcomm Hexagon LLVM C/C++ Compiler User Guide*.

3.3 Screen messages

The following example shows the screen messages generated by the Resource Analyzer.

```
C:>hexagon-analyzer-backend --elffile bootimg.pbn --aidfile
aidfile.txt -o ./outputs
Processing ELF file bootimg.pbn
Processing aid file aidfile.txt
Processing aid file
C:\Qualcomm\HEXAGON_Tools\5.0.05\qc\bin/libcAidFile.txt
Calling hexagon-objdump ...
Creating disassembly file ./outputs\bootimg.pbn.dis from bootimg.pbn
Please wait ... This could take several minutes with a large elf file.
hexagon-objdump complete.
Processing Disassembly file ./outputs\bootimg.pbn.dis
0% complete
18% complete
36% complete
55% complete
74% complete
94% complete
100% complete
Writing output file ./outputs\RA_FunctionStackSizes.csv
Writing output file ./outputs\RA_FunctionsByName.txt
Writing output file ./outputs\RA_WorstCaseCallPaths.txt
Writing output file ./outputs\RA_UnknownFunctions.txt
Number of functions in ELF file: 856
Number of memory objects in ELF file: 282
Backend Processing Complete
```

3.4 Errors and warnings

The Resource Analyzer outputs error messages when a failure occurs. Errors cannot be disabled.

The analyzer also generates the following warnings. All warnings can be enabled or disabled.

3.4.1 Could not compute R29 = for function name in file name

The analyzer can only interpret three types of operations on the stack pointer:

```
allocframe(#48)
r29 = add(#-48)
r29 = sub(#48)
```

The R29 alias SP is also supported.

The analyzer cannot interpret other manipulations of the stack pointer. For example:

```
r29 = memw(r23+#0)
r29 = r9
```

In these cases, the stack value of the function is marked as `INDETERMINATE`, and manual inspection is required to determine the function's actual stack value.

3.4.2 Missing @function declaration for function name in file name

The function does not have an `@function` declaration, which results in the function being categorized as an `STT_NOTYPE` in the ELF symbol table. Also, the profiler does not interpret `STT_NOTYPE` as a function.

All functions must have the following declaration:

```
.type funcname, @function
```

3.4.3 Missing .size directive for function name in file name

The function does not have a terminating `.size` directive, which results in the function being categorized as an `STT_NOTYPE` in the ELF symbol table with a size of zero. Also, the profiler does not interpret `STT_NOTYPE` as a function.

All functions must be terminated with the directive:

```
.size funcname, .-funcname
```

3.4.4 Missing @object declaration for object name in file name

The object does not have an `@object` declaration, which results in the object being categorized as an `STT_NOTYPE` in the ELF symbol table.

All memory objects must have the following declaration:

```
.type objectname, @object
```

3.4.5 Missing .size directive for object name in file name

The object does not have a terminating `.size` directive, which results in the object being categorized as an `STT_NOTYPE` in the ELF symbol table with a size of zero. Also, the profiler does not interpret `STT_NOTYPE` as a function.

All objects must be terminated with the directive:

```
.size objectcname, .-objectcname
```

3.4.6 Label name does not begin with .L in file name

Labels must begin with `.L`, otherwise they are categorized as an `STT_NOTYPE` item in the ELF symbol table.

3.4.7 Global collision for function name in file name

The analyzer found two global functions with the same name.

The analyzer forces the second instance of the function to have local scope. Any subsequent files that reference the function use the first function found.

3.4.8 Global collision for object name in file name

The analyzer found two global objects with the same name.

The analyzer forces the second instance of the object to have local scope. Any subsequent files that reference the object use the first object found.

3.4.9 Recursion encountered at function name in file name

This function has already been encountered at a higher node in this branch.

This node and the entire call path are marked as `RECURSIVE`. The cumulative stack value cannot be calculated.

3.5 Output files

The Resource Analyzer writes the resource usage information to text files. The following sections describe these files.

NOTE: The cumulative stack value of a function is only accurate if the branch type of that function is Closed.

3.5.1 List functions by name

The analyzer outputs a file that lists all the functions alphabetically with their attributes and call lists.

Output file name: `FunctionsByName.txt`

3.5.2 List functions by size

The analyzer outputs a file that lists all the functions sorted by largest cumulative stack size with their attributes and call lists.

Output file name: `FunctionsBySize.txt`

3.5.3 List function call paths

The analyzer outputs a file that lists all the functions sorted by largest cumulative stack size with a list of the largest call path.

Output file name: `FunctionsCallPaths.txt`

3.5.4 List unknown functions

The analyzer outputs a file that lists all the unknown functions, which means there were calls to functions not found in the database.

Output file name: `UnknownFunctions.txt`

3.5.5 File format

The output files are generated as text files in an easy-to-read format. For example:

```
   Function Name: ThreadMain
    Mangled Name: ThreadMain
     Branch Type: Closed
           Scope: Global
           Stack: 48 bytes
Cumulative Stack: 1128 bytes
           Calls: Called function         Cumulative Stack, Branch Type
                  __save_r16_through_r23                 0, Closed
                  qurt_sem_down                         16, Closed
                  printf                               904, Closed
                  qurt_pipe_send                      1032, Closed
                  __restore_r16_through_r23_and_deallocframe_before_tailcall    0, Closed
                  qurt_thread_exit                    1080, Closed
                  qurt_pipe_receive                   1032, Closed
                  qurt_barrier_init                      0, Closed
                  qurt_sem_init_val                      0, Closed
                  qurt_pipe_init                        16, Closed
                  qurt_sem_add                           0, Closed
```

The `RA_FunctionStackSizes.csv` file is better suited for postprocessing with a script. The information in this file has the following data format:

```
function_name, cumulative_stack_size, branch_type
```

For example:

```
ThreadMain,1128,Closed
_Mbtowc,1128,Closed
fputc,1128,Closed
_pthread_stub,1120,Closed
pthread_mutex_lock,1120,Closed
pthread_mutex_trylock,1120,Closed
_Fwprep,1112,Closed
pthread_cond_wait,1112,Closed
_Getmbcurmax,1104,Closed
_Getpctype,1104,Closed
__register_frame_info_bases,1104,Closed
__registerx,1104,Closed
pthread_cond_broadcast,1104,Closed
```

3.6 Aiding files

The Resource Analyzer optionally accepts (with the `--aidfile` option) an aiding file that provides information on missing resources. You can also use aiding files to override certain resource information (such as the function stack size and call list).

Each aiding file has the following properties:

- The file does not contain branches; instead, it contains only nodes that are treated as leafs.

  The stack size of a node in the file is the cumulative stack size for that node.
- The file can include comments that are preceded with the double forward slash. No other commenting styles are supported.
- For functions, nodes in an aiding file have the following data format:

  ```
  code; symbol; size; scope
  ```

  Where:

  `code` – Informs the analyzer that this is a function node

  `symbol` – The function name (for C++, it must be the mangled name)

  `size` – The cumulative stack size of the node

  `scope` – The scope of the function
- For memory blocks, nodes in an aiding file have the following data format:

  ```
  data; symbol; size; scope
  ```

  Where:

  `data` – Informs the analyzer that this is a memory block

  `symbol` – The memory block name

  `size` – The memory block size

  `scope` – The scope of the function
- The scope of a node in the aiding file has one of the following values.
  - 0 – Local
  - 1 – Global
  - 2 – Weak
  - 3 – Override
- The analyzer supports the following delimiters:

  ```
  <comma> <space> <tab> <semicolon>
  ```

  The following lines are valid:

  ```
  code name, 128, 1// comment
  code;_Znwj;300, 1
  ```

Example aiding file

```
// Format:
// code;name;stacksize;scope;
// data;name;memsize;scope;
code;function1;288;1;
code;function2;144;3;
data;memobject1;4;2;
data;memobject2;400;3;
```

3.7 Root specification files

The Resource Analyzer optionally accepts (with the `--rootfile` option) a file that identifies the default program entry points. These files are known as *root specification* files.

Each root specification file is a text file that contains a list of function names, with one name per line. The # character indicates a comment and is ignored by the analyzer.

Example of a root specification file:

```
threadmain
main
test
```

3.8 Handling recursive and indeterminate functions

While recursion can be caused by a function calling itself, it is more often caused by a function tree that loops back on itself. The best way to resolve recursion is to rewrite the code to be non-recursive; however this may not always be possible.

The Resource Analyzer provides a mechanism for breaking recursion through the use of an aiding file (Section 3.6). The aiding file allows you to override any function's cumulative stack value and remove its call list, thus breaking the recursive chain.

Indeterminate functions are caused by direct manipulation of the stack pointer with run-time arguments. The cumulative stack value of an indeterminate function must be considered as *at least*. You can use the aiding file to override the indeterminate state of the function.

Perform overriding by setting the scope value to 3 in the aiding file (for details, see Section 3.6).

4 Analysis methodology

4.1 Function tree definitions

A function tree has a root, branches, and leaves. The root of a tree is a function, and function calls from the root form the branches. A function that makes no calls is considered a leaf.

- A node is a function, and it can be a root, a branch, or a leaf.
- The root is considered to be the top of the tree, and the leaves are the bottom. Therefore:
  - A node that is up or above means traversing towards the root.
  - A node that is down or below means traversing towards the leaves.

4.1.1 Function declarations

The analyzer interprets the following keywords and builds function trees accordingly.

NOTE: All keywords are case insensitive.

4.1.1.1 Scope declarations

- `.global` symbol
- `.globl` symbol
- `.weak` symbol

Functions without a scope are treated as local.

4.1.1.2 Function declarations

- The analyzer interprets the following as a function:

  ```
  .type symbol, @function
  ```

- The analyzer supports only three MACRO declarations:

  `FUNCTION_BEGIN`

  `HEXAGON_OPT_FUNC_BEGIN`

  `FALLTHROUGH_TAIL_CALL` (terminates last and declares next)

- Functions defined in any other manner are not identified.
- Functions are terminated with one of the following:

```
.size symbol
FUNCTION_END
HEXAGON_OPT_FUNCTION_FINISH
```

4.1.1.3 Function calls

- Normal function call:

```
call symbol
```

- Jump to function is treated as a function call:

```
jump symbol
```

- The analyzer has type lists to differentiate between functions and addresses:

```
jump .L127 vs. jump .LTHUNK0
```

4.1.2 Stack allocation declarations

The analyzer calculates stack requirements with the following rules:

- The stack requirement of any node in a tree is calculated to be the cumulative stack requirement of a called function that contains the largest value plus the value of the node.
- Stack consumption keywords:
  - The following examples add to the function stack:

```
ALLOCFRAME(#80)
SP = add(SP,#-80)
```

  - Code that subtracts from stack is ignored:

```
R29 = add(R29,#80)
```

  - The analyzer adds 8 to stack requirements when `ALLOCFRAME` is parsed.
- The analyzer interprets everything between the `#` and `)` characters to be a number.

  The following examples are valid:

```
ALLOCFRAME(#(2+46))
ALLOCFRAME(#(((2*8)+(4*4))/2))
SP = ADD(SP,#-((4+3)*9))
```

4.1.3 Branch and node types

The analyzer categorizes nodes and branches to be one of the following types.

4.1.3.1 Closed

All branches terminate in found leafs, and the stack size of all nodes is known.

Only closed branches have accurate cumulative stack values.

4.1.3.2 Open

If a branch terminates in leafs that are not found, the entire branch is open.

4.1.3.3 Indeterminate

If the stack size of any node in a branch cannot be determined, the entire branch is indeterminate.

This type is typically caused by assembly code that directly manipulates stack pointer register R29.

4.1.3.4 Recursive

If a function calls any node above the current node, the branch is recursive.

The cumulative stack value of a recursive function is set to the cumulative stack value of the first called function it processes. The actual value is infinity.

4.1.4 Function name aliasing with .set keyword

The analyzer handles the `.set` keyword by keeping a list of the aliases.

- The analyzer maintains an alias list while processing a file.
- The call lists are resolved to actual function names, not the alias.

4.2 Memory information

The analyzer collects memory requirements information from the assembly files and processes the object keyword to identify memory:

```
.type symbol, @object
.size symbol, memsize
```