Profiling Nios II Systems

AN-391-3.0 Application Note

This application note describes the methods to measure the performance of a Nios® II system with the GNU profiler (**nios2-elf-gprof)**, the performance counter component, and the timestamp interval timer component. This application note also includes two tutorials to measure performance in the Altera® Nios II Software Build Tools (SBT) development flow.

Requirements

You must be familiar with the Nios II SBT development flow for Nios II systems, including the Quartus® II software and Qsys to use the tutorials.

Obtaining the Hardware Design

The tutorials in this application note work with the Nios II Ethernet Standard Design Example.

To use the design example, unzip the **.zip** for your development kit to a working directory on your system.

This application note refers the software example directory as <*project_directory*>.

Obtaining the Software Examples

To obtain the software examples for this application note, follow these steps:

1. Download the **profiler_software_examples.zip**.
2. Unzip the **profiler_software_examples.zip** to <*project_directory*> in your system.

   This application note refers the directory as <*profiler_software_examples*>.

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Tools

You can use the GNU profiler without making any hardware changes to your Nios II system. This tool directs the compiler to add calls to profiler library functions into your application code.

The performance counter component and the timestamp component are minimally intrusive hardware methods for measuring the performance of a Nios II system. This application note describes and compares the two components. To use these methods, you add the hardware components to your system, and you add macro invocations to your source code to start and stop the components. The hardware components perform the measurements.

Compiler speed optimizations affect functions to widely varying degrees. Compiler size optimizations also affect functions in different ways. These differences impact the cache usage and the resource contention, which can change the relative start times and therefore increase the execution times of functions. For these reasons, you must optimize your code with the `-O3` compiler switch, and then perform profiling on the code to gain the most insight on how to improve an application in its final form.

The tutorials use three tools to measure the performance of a Nios II system, as described in the following sections:

- GNU Profiler
- Altera Performance Counter
- High-Resolution Timer

In addition, the program counter trace collection tool is available for some Nios II processors. However, the tutorials do not use this tool.

You use the GNU profiler to identify the areas of code that consume the most CPU time, and a performance counter or a timer component to analyze functional bottlenecks.

GNU Profiler

You must make minimal changes to the source code to take measurements for analysis with the GNU profiler. To implement the required changes, follow these steps:

1. In the Nios II SBT, enable the GNU profiler in your project by turning on the `hal.enable_gprof` and `hal.enable_exit` board support package (BSP) settings.

   ☞ If you use the Nios II SBT for Eclipse, the software enables `hal.enable_exit` by default.

2. Verify that your `main()` function returns.

   ☞ When `main()` calls `return()` or terminates, `alt_main()` calls `exit()` as appropriate for profiling. The `exit()` function runs the `BREAK 2` instruction, which causes the profiling data to write to the **gmon.out** on the host computer.

3. Rebuild the BSP and the application project.

Altera Performance Counter

A performance counter is a block of counters in the hardware that measures the execution time of the code sections that you choose. A performance counter component can track up to seven code sections. By default, the component tracks three code sections. A pair of counters tracks each code section:

- *Time*—A 64-bit time (clock tick) counter that counts the number of clock ticks during code section runs.
- *Occurrences*—A 32-bit event counter that counts the number of times the code section runs.

☞ You can change the maximum number of measured code sections by editing the performance counter component in Qsys.

These counters enable you to measure the execution time of the designated sections of C/C++ code. Macros enable you to mark the start and the end of the code sections in your program. The performance counter component has up to seven pairs of counters, supporting as many as seven measured sections of C/C++ code. You must add macros to your code at the start and end of each measured section. An additional, built-in pair of counters aggregates the individual code section counters, enabling you to measure each section as a fraction of a larger program.

You can use performance counters for analyzing determinism and other runtime issues.

☞ The performance counter component occupies a substantial number of logic elements (LEs) on your device, and requires software implementation to obtain performance measurements.

High-Resolution Timer

A high-resolution timer, in contrast to a performance counter component, does not use a large number of LEs on your device, and does not require heavy implementation of every function call in your code to obtain performance measurements. Timers require explicit read calls in the sections of the source code that you want to measure, so their use is better suited for pinpointing the performance issues in a program. You must implement the source code manually; however, because this implementation is less pervasive, therefore, this implementation is also less intrusive. Unlike the performance counter macros, the timer requires many more processor cycles to make two function calls; one to read the time at the beginning of a measured section, and one to read the time at the end.

Program Counter Trace Information

The Nios II processor can generate complete and accurate program counter trace information. However, the GNU profiler does not use this information. To generate this information, you must have a Nios II processor configured with a JTAG debug module of level 3 or greater. The level 3 JTAG debug module creates on-chip trace data. You can capture approximately a dozen instructions in the on-chip trace buffer.

You can obtain a much larger trace by configuring a Nios II core with a level 4 JTAG debug module to generate off-chip trace information; however, you need a First Silicon Solutions, Inc. (FS2) or Lauterbach Datentechnik GmBH (Lauterbach) (www.lauterbach.com) hardware to collect this off-chip trace data.

For more information about the Lauterbach hardware, refer to "Debuggers" in the *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*.

Using the GNU Profiler to Measure Code Performance

The following sections explain the advantages and limitations of using the GNU profiler for performance analysis. A tutorial demonstrates the use of the GNU profiler to collect and analyze performance data.

GNU Profiler Advantages

The major advantage to measuring performance with the GNU profiler is that the GNU profiler provides an overview of the entire system. Although the GNU profiler adds some overhead, the GNU profiler distributes this overhead throughout the system evenly. The functions the GNU profiler identifies as consuming the most processor time also consume the most processor time when you run the application at full speed without profiler implementation.

GNU Profiler Limitations

Adding instructions to each function call for use by the GNU profiler affects the behavior of the code in the following ways:

- Each function is slightly larger due to the additional function call to collect profiling information.
- The entry and the exit time of each function due to profiling information collection.
- The instruction-cache misses are higher because the profiling function is in the instruction cache memory.
- Memory that records the profiling data can change the behavior of the data cache.

These effects can mask the time sensitive issues that you are trying to uncover through profiling.

The GNU profiler determines the percentage of time spent in each function by interpolation, based on periodic samplings of the program counter. The GNU profiler ties the periodic samples to the timer tick of the system clock. The GNU profiler can take samples only when you enable interrupts, and therefore cannot record the processor cycles spent in interrupt routines.

The GNU profiler cannot profile individual functions. You can use the GNU profiler to profile the entire system, or not at all.

The profiling data is a sampling of the program counter at the resolution of the system timer tick. Therefore, the profiling data provides estimation, not an exact representation, of the processor time spent in different functions. You can improve the statistical significance of the sampling by increasing the frequency of the system timer tick. However, increasing the frequency of the tick increases the time spent recording samples, which in turn affects the integrity of the measurement.

To use the GNU profiler successfully with your custom hardware design, you must ensure that your design includes a system clock timer. The GNU profiler requires this component to produce proper output.

Software Considerations

The GNU profiler implements your source code with functions to track processor usage.

Profiler Mechanics

You enable the GNU profiler by turning on the `hal.enable_gprof` switch in the scripts to generate the BSP. Turning on this switch automatically turns on the `-pg` compiler switch and then links the profiling library code in the `altera_nios2` software component with the BSP. This code counts the number of calls to each profiled function.

The `-pg` compiler option forces the compiler to insert a call to the `mcount()` function (located in the file **altera_nios2/HAL/src/alt_mcount.S**) at the beginning of every function call. The calls to `mcount()` track every dynamic parent and child function call relationship to enable the construction of a call graph. The option also installs `nios2_pcsample()` function (located in the file **altera_nios2/HAL/src/alt_gmon.c**) that samples the foreground program counter at every system clock interrupt. When the program executes, the GNU profiler collects data on the host of the **gmon.out**. The **nios2-elf-gprof** utility can read this file and display profiling information about the program.

The profiling code operates on the target by performing the following steps:

1. The Compiler implements function prologues with a call to `mcount()` to enable the Compiler to determine the function call graph. The GNU profiler documentation refers to this data as the function call arc data.
2. The timer interrupt handler registers an alarm to capture information about the foreground function (histogram data) that executes when the alarm triggers.
3. The heap allocates a target memory to store the profiling data.
4. When your code exits with a `BREAK 2` instruction, the **nios2-download** utility copies the profiling data from the target to the host.

The **nios2-elf-gprof** utility requires the function call arc data and the histogram data to work correctly.

For more information about the GNU profiler, refer to the Nios II GNU profiler documentation, included with the GCC documentation, available on the Nios II Embedded Design Suite Support page of the Altera website.

Profiler Overhead

Using the GNU profiler impacts memory and processor cycles.

Memory

The impact of the profiling information on the `.text` section size is proportional to the number of small functions in the application. The code overhead—the size of the `.text` section—increases when the GNU profiler enables profiling, due to the addition of the `nios2_pcsample()` and `mcount()` functions. The GNU profiler implements the system timer with a call to `nios2_pcsample()`, and implements every function with a call to `mcount()`. The `.text` section increases by the additional function calls and by the sizes of these two functions.

To view the impact on the `.text` section, you can compare the sizes of the `.text` sections in the **.objdump**.

The GNU profiler uses buckets to store data on the heap during profiling. Each bucket is two bytes in size. Each bucket holds samples for 32 bytes of code in the `.text` section. The total number of profiler buckets allocated from the heap is when you divide the size of the `.text` section by 32. The heap memory that the GNU profiler buckets consume is therefore:

((`.text` section size) / 32) × 2 bytes

The GNU profiler measures all functions in the object code that the GNU profiler compiles with profiling information. This set of functions includes the library functions, which include the run-time library and the BSP.

Processor Cycles

The GNU profiler tracks each individual function with a call to `mcount()`. Therefore, if the application code contains many small functions, the impact of the GNU profiler on processor time is larger. However, the resolution of the profiled data is higher. To calculate the additional processor time consumed by profiling with `mcount()`, multiply the amount of time that the processor requires to execute `mcount()` by the number of run-time function calls in your application run.

On every clock tick, the processor calls the `nios2_pcsample()` function. To calculate the required additional processor time to perform profiling with `nios2_pcsample()`, multiply the time the processor requires to execute this function by the number of clock ticks that your application requires, which includes the time the `mcount()` calls and execution requires.

To calculate the number of additional processor cycles used for profiling, add the overhead you calculated for all the calls to `mcount()` to the overhead you calculated for all the calls to `nios2_pcsample()`.

Hardware Considerations

The GNU profiler requires only a system timer. If your Nios II hardware design includes a system timer, you do not need to change your design.

Tutorial: Using the GNU Profiler

For demonstration purposes, this tutorial uses the Nios II Ethernet Standard design example for the Nios II Embedded Evaluation Kit, Cyclone III Edition (NEEK) development kit. You could use other similar design examples which target your development kit.

To configure your device with the design example, follow these steps:

1. Start the Quartus II software, version 11.0 or later.
2. On the File menu, click **Open Project**.
3. Open **niosii_ethernet_standard_3c25.qpf**.
4. On the Tools menu, click **Programmer**.
5. Click **Start** to download the SRAM Object File (**.sof**) to your device.

If the software disabled the **Start** button, or the **Hardware Setup** field does not list the USB-Blaster™ cable, refer to the *Introduction to the Quartus II Software* manual for more details on the Programmer tool.

Profiler Example with the Nios II Command Line

Creating the Profiler Software Example

To create the profiler_gnu software project in the Nios II command-line flow, follow these steps:

1. Open a Nios II command shell by executing one of the following steps, depending on your environment:
   - In the Windows operating system, on the Start menu, point to **Programs** > **Altera** > **Nios II EDS** <*version*>, and click **Nios II** <*version*> **Command Shell**.
   - In the Linux operating system, in a command shell, change directories to <*Nios II EDS install path*>, and type the command `./sdk_shell`.
2. Change to the directory <*profiler_software_examples*>**/app/profiler_gnu**
3. Create and build the application with the **create-this-app** script, by typing the following command:

```
./create-this-app ↵
```

The **create-this-app** script runs the **create-this-bsp** script, which reads settings from the **parameter_definition.tcl** in <*profiler_software_examples*>**/bsp/hal_profiler_gnu**. This Tcl file contains the following lines:

```
set_setting hal.enable_gprof true
set_setting hal.enable_exit true
```

The first setting enables the GNU profiler, and the second setting enables the `alt_main()` function to call `exit()` following `main()`.

Running the Profiler Software Example

To run the application and collect the GNU profiler data, follow these steps:

1. Open a second Nios II command shell.
2. In the second shell, open a **nios2-terminal** session by typing the following command:

   ```
   nios2-terminal ↵
   ```

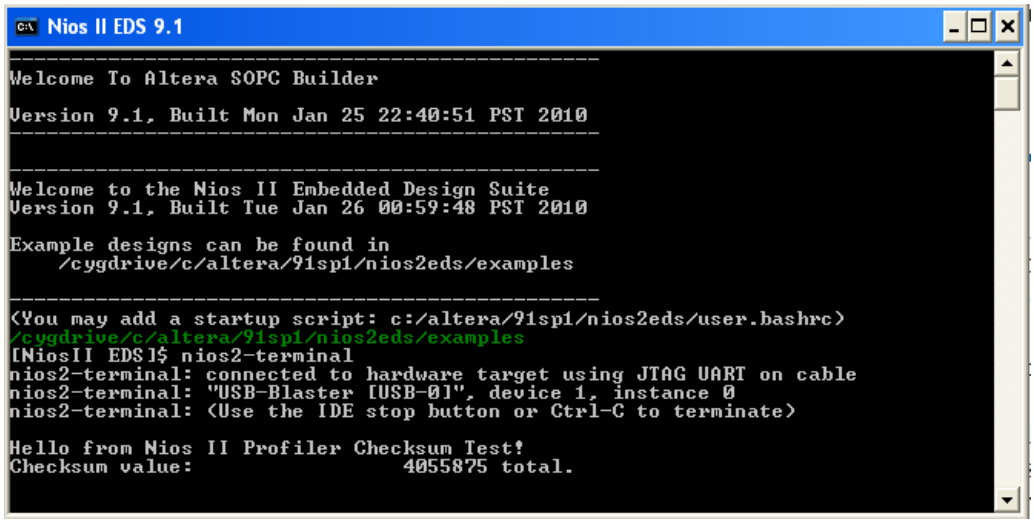
3. In your original Nios II command shell, download the .**elf** to the development board, run your design, and write the GNU profiler data to the **gmon.out**, by typing the following command:

   ```
   nios2-download -g --write-gmon gmon.out *.elf ↵
   ```

   The GNU profiler collects data while the application runs, and then writes the data to the **gmon.out** when the application calls the `exit()` function. Figure 1 shows an example of the GNU profiler output in the Nios II command shell.

4. Exit **nios2-terminal** by typing `control-C`.

Figure 1. GNU Profiler Output on Nios II Command Shell



Creating the GNU Profiler Report

When you run your project, your project creates the **gmon.out**. You must format this file to a readable format. To format this file, follow these steps:

1. In the original Nios II command shell, change your directory to *<profiler_software_examples>***/app/profiler_gnu**.
2. Type the following command:

   ```
   nios2-elf-gprof profiler_gnu.elf gmon.out > report.txt ↵
   ```

   This command generates a flat profile report and a call graph, which you can view in the **report.txt**.

3. Use any text editor to view the **report.txt**.

For more information about the GNU profiler report, refer to "Analyzing the GNU Profiler Report" on page 10 .

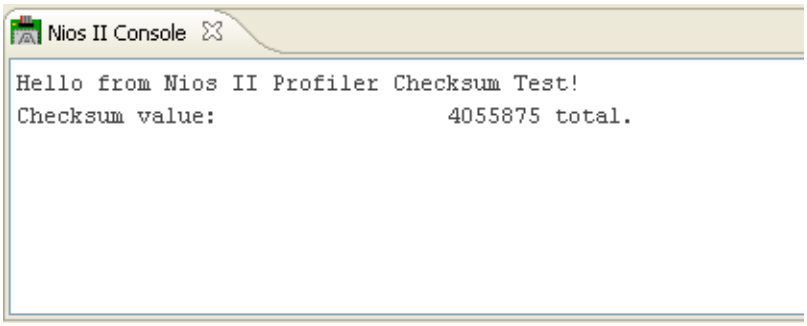
Profiler Example with Nios II SBT for Eclipse

Creating and Running the Profiler Software Example

1. Start the Nios II SBT for Eclipse.
2. Under File menu, point to **New**, and click **Nios II Application and BSP from template**.
3. Set **SOPC Information File name** by browsing to locate the SOPC Information File (**.sopcinfo**) in <*project_directory*>.
4. Name your project, such as profiler_gnu.
5. Under **Templates**, select **Blank Project**.
6. Click **Finish** to create your new project.
7. Locate the <*profiler_software_examples*>**/eclipse_source_files/profiler_gnu** folder and copy all the files in this directory. In Nios II SBT for Eclipse, right click on **profiler_gnu** in Project Explorer view and select **Paste**.
8. Right click your project in the Project Explorer view, point to **Nios II** and click **BSP Editor**.
9. In the Nios II BSP Editor, turn on `hal.enable_gprof` to enable the GNU profiler in your project.
10. Generate your BSP project and exit.
11. Right click your project in the Project Explorer view and then click **Build Project**.
12. To download and run the profiler_gnu software, right click your project, point to **Run As**, and then click **Nios II Hardware**.

Figure 2 shows the output of the software to the Nios II console.

Figure 2. Nios II Console After Running profiler_gnu



Viewing the GNU Profiler Report

The software creates the **gmon.out** in your project folder, which you can view in the Project Explorer view of the Nios II SBT for Eclipse. If the **gmon.out** does not appear, right click on your project and select **Refresh**. When you open **gmon.out**, the Nios II SBT for Eclipse switches to the Profiling view, in which you can view the report. For more information about the GNU profiler report, refer to "Analyzing the GNU Profiler Report".

Analyzing the GNU Profiler Report

The information in this section is applicable to the GNU profiler report that the command line or the Nios II SBT for Eclipse generates.

The GNU profiler report contains information in the following formats:

- The **flat profile** portion of the report identifies the child functions in the order in which they consume processing time.
- The **call graph** portion of the report describes the call tree of the program sorted by the total amount of time spent in each function and its children. Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called, with exceptions and conditions detailed further in the report itself and the GNU profiler documentation.

For more information, refer to the Nios II GNU profiler documentation, with the GCC documentation, available at the Nios II Embedded Design Suite Support page.

Example 1 shows the GNU profiler report excerpts from the previous tutorial. In Example 1, the flat profile shows that the `checksum_test_routine()` function call consumed 79.19% of the processing time during the execution.

The granularity statement in the call graph report states that the report covers 2.55 seconds (2550 milliseconds). The Nios II timer (`sys_clk_timer`) has a 10 millisecond timer. The GNU profiler calls the timer interrupt once at the beginning, before a full clock period elapsed, and once every 10 milliseconds thereafter. A precise report, therefore, would show that the GNU profiler calls the timer interrupt handler 255 times. Index[13] shows that the GNU profiler calls `alt_avalon_timer_sc_irq()` 256 times, which is in the precision range of this measurement method.

Note that the result you see may vary from Example 1.

Example 1. Flat Profile and Call Graph Example

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 79.19      2.02     2.02        1     2.02     2.03  checksum_test_routine
 18.01      2.48     0.46        1     0.46     0.46  alt_busy_sleep

        .
        .
        .

            Call graph (explanation follows)


granularity: each sample hit covers 32 byte(s) for 0.39% of 2.55 seconds

index      % time    self    children    called     name
                0.00    0.00           273/273         alt_irq_entry [106]
[13]    0.0    0.00    0.00              273         alt_irq_handler [13]
                0.00    0.00           256/256         alt_avalon_timer_sc_irq [14]
                0.00    0.00            17/17          altera_avalon_jtag_uart_irq [17]

        .
        .
        .
```

Using Performance Counter and Timer Components

After the GNU profiler identifies areas of code that consume the most processor time, a performance counter or a timer component can further analyze these functional bottlenecks.

The following sections describe the advantages and limitations of using performance counters and timers for performance analysis. A tutorial demonstrates the use of performance counters and timers to collect and analyze performance data.

Performance Counter Advantages

Performance counters are the only mechanism available with the Nios II development kits that provide measurements with little intrusion. You can use efficient macros to start and stop the measurement for each measured section. A performance counter is an order of magnitude faster than the GNU profiler. The only less intrusive way to collect measurement data would be a completely hardware-based solution, such as a logic analyzer configured with triggers on particular bus addresses.

Timer Advantages

Unlike the performance counter, which can track only seven sections of code simultaneously, the timer has no such limit. You can read the timer 1,000 times and store the timer in 1,000 different variables as a start time for a section. Then, you can compare the timer to 1,000 end timer readings. The only practical limiting factors are memory consumption, processor overhead, and complexity.

Performance Counter and Timer Hardware Considerations

One disadvantage to measuring performance with a performance counter is the size of the counter. The performance counter component consumes a large number of LEs on your device.

On a 3C120 device, a single performance counter component with three section counters defined in a modified standard hardware design consumes 671 logic cells (LCs), and 420 LC registers. In the same design, a single performance counter defined with seven section counters consumes 1339 logic cells and 808 LC registers. The resource usage of the performance counter component is nearly identical on all devices.

Remove the performance counter from the final version of your system to save resources.

The timer consumes hardware resources, although substantially less than a performance counter. The timer also introduces an additional interrupt source in the system that impacts interrupt latency.

Adding performance counters and timers can reduce f_{MAX}.

Performance Counter and Timer Software Considerations

A common disadvantage of performance counters and timers is the lack of context awareness. If a timer interrupt occurs during the measurement of a section of code, performance counters and timers add the time taken by the processor to process the timer interrupt to the total measurement time. This effect occurs for simple interrupts and multithreading context switching, although this effect occurs more in a multithreaded system. Many threads or interrupt service routines might execute while you measure the section of code, resulting in a very large, skewed measurement. The resulting measurement distortion is unpredictable, and has no correlation with the behavior of the code section you are attempting to measure.

To avoid context switch impacts, most multithreaded operating systems have a system call to temporarily lock the scheduler. Alternatively, you can disable interrupts to avoid context switches during section measurement.

Disabling interrupts or locking the scheduler affects the behavior of your system, so you must use these techniques only as a last resort.

Performance Counter Software Considerations

You must use the PERF_BEGIN and PERF_END performance counter macros to record the beginning and ending times of each measured section.

PERF_BEGIN and PERF_END are single writes to the performance counter component. These macros are very efficient, requiring only two or three machine instructions.

Example 2 shows the PERF_BEGIN and PERF_END performance counter macros in **altera_avalon_performance_counter.h**:

Example 2. PERF_BEGIN and PERF_END Performance Counter Macros in altera_avalon_performance_counter.h

```
#define PERF_BEGIN(p,n) IOWR((p),(((n)*4)+1),0)

#define PERF_END(p,n) IOWR((p),(((n)*4) ),0)
```

The Global Counter

The performance counter component contains several counters. You can configure the number of measured sections in Qsys. You have one pair of counters for each measured section, as described in "Altera Performance Counter" on page 3. In addition, the performance counter component always has a global counter.

The global counter measures the total time of the measurement. When you stop the global counter, other counters do not run. The PERF_START_MEASURING and PERF_STOP_MEASURING macros control the global counter.

Do not attempt to manipulate the global counter in any other way.

For more information about performance counters, refer to the *Performance Counter Core* chapter in the *Embedded Peripherals IP User Guide*.

Hardware Considerations

Performance counters and timestamp interval timers are Qsys components. When you add one to an existing system, you must regenerate the Qsys system and recompile the **.sof** in the Quartus II software. Timers and performance counters can eventually overflow, such as any hardware counter.

Tutorial: Using Performance Counters and Timers

This tutorial demonstrates the use of performance counters and timestamp interval timers to measure the performance of a Nios II system more precisely than is possible with the GNU profiler, by identifying the sections of code that use the most processor time.

This tutorial uses the same NEEK design as the GNU profiler tutorial. This design has an interval timer and a performance counter. You can change the timer interval and the number of sections that the performance counter measures.

Modifying the Nios II Hardware Design

You must modify the Nios II Ethernet Standard design example for this tutorial. To modify the Nios II Ethernet Standard design example, follow these steps:

1. In Quartus II software, on the Tools menu, click **Qsys**.
2. In <*project_directory*>, click **peripheral_system.qsys**.
3. Right click the **high_res_timer** module and then click **Edit**.
4. Under **Timeout period**, set the interval time **Period** to `1` and the units to **us** (microseconds).
5. Click **Finish**.
6. On the File menu, click **Save**.
7. The Nios II Ethernet Standard design example is a hierarchal based design. To generate the system, on the File menu, click **Open**, and then select **eth_std_main_system.qsys**.
8. Click the **Generation** tab.
9. Turn on the **Create HDL design files for synthesis and Create block symbol file (.bsf)** options.
10. Ensure that the Output Directory path is <*project_directory*>**/eth_std_main_system**.
11. Click **Generate**. Save the system if the software prompts you to do so.
12. Exit Qsys when generation is complete.
13. To generate the .**sof,** in the Quartus II software, on the Processing menu, click **Start Compilation**.
14. Click **OK** when the following message appears:

```
Full Compilation was successful
```

Programming the Hardware Design to Your Device

After compiling your modified hardware design, you can program the hardware design to your device. To do so, follow these steps:

1. On the Tools menu, click **Programmer**.
2. Click **Start** to download the .**sof** to your device.

If the software disables the **Start** button, or the **Hardware Setup** field does not list the USB-Blaster cable, refer to the *Introduction to the Quartus II Software* manual for more details on the Programmer tool.

Performance Counter Example with the Nios II Command Line

This section describes how to create and run the performance counter software example with the Nios II command line.

Creating the Performance Counter Software Example

To create the **profiler_performance_counter** software project in the Nios II software build flow, follow these steps:

1. Open a Nios II command shell as described in "Creating the Profiler Software Example" on page 7.
2. Change to the <*profiler_software_examples*>**/app/profiler_performance_counter** directory.
3. Create and build the application by typing the following command:

   ```
   ./create-this-app ↵
   ```

   The **create-this-app** script runs the **create-this-bsp** script, which reads settings from the **parameter_definition.tcl** in <*profiler_software_examples*>**/bsp/hal_profiler_performance_counter**. This Tcl file contains the following lines:

   ```
   set_setting hal.sys_clk_timer peripheral_subsystem_sys_clk_timer
   set_setting hal.timestamp_timer peripheral_subsystem_high_res_timer
   set_setting hal.enable_gprof true
   set_setting hal.enable_exit true
   ```

   The first two lines set the system clock timer and timestamp timer to the corresponding timers in the Qsys system.

   The third line enables the GNU profiler, and the last line enable the `alt_main()` function to call `exit()` following `main()`.

Running the Performance Counter Software Example

To run the application and collect the GNU profiler data, follow these steps:

1. Open a second Nios II command shell.
2. In the second shell, open a **nios2-terminal** session by typing the following command:

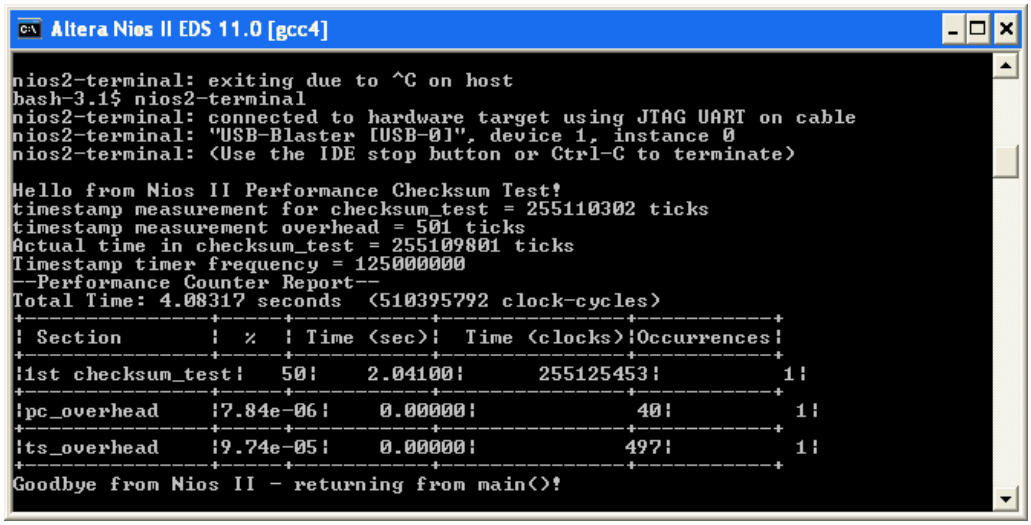
   ```
   nios2-terminal ↵
   ```

3. In your original Nios II command shell, run the program by typing the following command:

   ```
   nios2-download -g *.elf ↵
   ```

Figure 3 shows an example of the output that appears in the Nios II command shell. Your output might vary. For more information, refer to "Analyzing the Performance Counter Report".

Figure 3. Performance Counter Report on Nios II Command Shell



Performance Counter Example with Nios II SBT for Eclipse

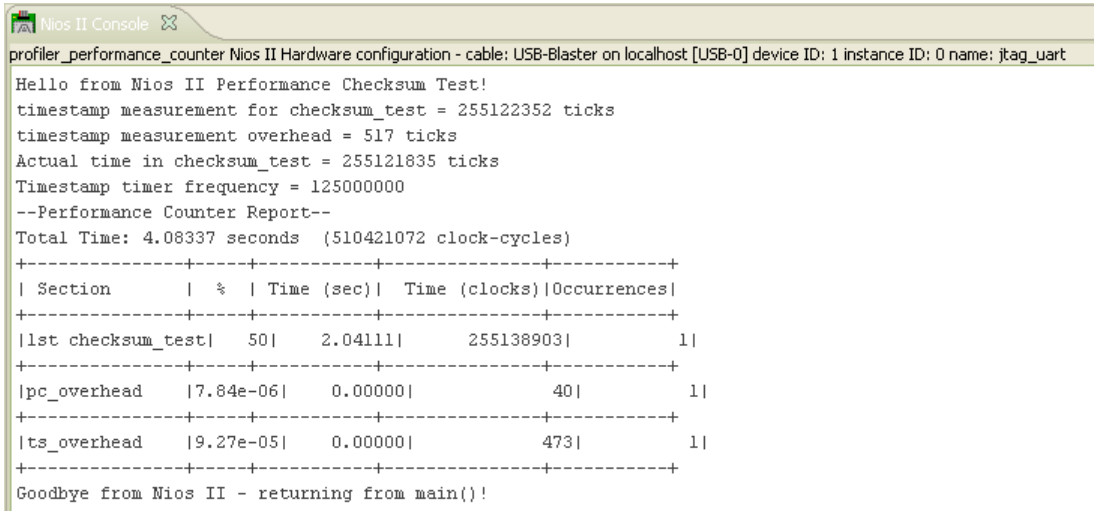
This section describes how to create and run the **profiler_performance_counter** software example with the Nios II SBT for Eclipse.

1. Start the Nios II SBT for Eclipse.
2. Under File menu, point to **New**, and click **Nios II Application and BSP from template**.
3. Set **SOPC Information File name** by browsing to the <*project_directory*> directory and selecting the .**sopcinfo**.
4. Give your project a name, for example **profiler_performance_counter**.
5. Under **Templates**, select **Blank Project**.
6. Click **Finish** to create your new project.
7. Locate the <*profiler_software_examples*>**/eclipse_source_files/profiler_performance_counter** folder, and copy all the files in this directory. In Nios II SBT for Eclipse, right click on profiler_gnu in Project Explorer view and select **Paste**.
8. Right click your project in the Project Explorer view, point to **Nios II** and click **BSP Editor**.
9. In the Nios II BSP Editor, turn on `hal.enable_gprof` to enable the GNU profiler in your project.
10. Set the `hal.sys_clk_timer` to the **peripheral_subsystem_sys_clk_timer** component.
11. Set `hal.timestamp_timer` to the **peripheral_subsystem_high_res_timer** component.

12. Generate your BSP project and exit.
13. Right click your project in the Project Explorer view, point to **Build Project**.
14. To run the **profiler_performance_counter** software, right click your application project, point to **Run As** and click **Nios II Hardware**.

Figure 4 shows the Nios II Console output after running **profiler_performance_counter**. The data are similar to the command-line example in Figure 3. For more information, refer to "Analyzing the Performance Counter Report".

Figure 4. Performance Counter Report on Nios II Console



Analyzing the Performance Counter Report

The information in this section is applicable to the performance counter report that the command line or the Nios II SBT for Eclipse generates.

`pc_overhead` is the performance counter component overhead due to a single call to the `PERF_BEGIN` macro. This number includes the overhead of executing the `PERF_BEGIN` macro and the corresponding `PERF_END` macro for this measured section.

`ts_overhead` is the timestamp overhead—the overhead of a single function call to read the timer. This number includes the performance counter overhead to implement the measurement.

Conclusion

The Nios II development environment provides several tools to analyze the performance of your project. The software-only GNU profiler approach adds minimal overhead. To analyze deterministic real-time performance issues, you can use a hardware timer or a performance counter. To choose the best tool for your task, consider the problem that you are solving.

Troubleshooting

The following sections describe several problems that might occur, and suggest ways to troubleshoot the problems.

nios2-elf-gprof –annotated-source Switch Has No Effect

The profiler does not track the `basic-block-count` information, so switches (such as the –`annotated-source` switch) do not work.

Writing to the Registers of a Nonexistent Section Counter

The performance counter report in Example 3 shows what happens when you attempt to use a nonexistent section counter of the performance counter component.

Example 3. Result of Using a Nonexistent Section Counter

```
--Performance Counter Report--
Total Time: 5.78751 seconds (289375582 clock-cycles)
+--------------------+--------+-------------+---------------+-----------+
|      Section       |   %    | Time (sec)  | Time (clocks) |Occurrences|
+--------------------+--------+-------------+---------------+-----------+
|sleep_tests         |    49.4|      2.86162|      143081026|          1|
+--------------------+--------+-------------+---------------+-----------+
|perf_begin_overhead | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|timestamp_overhead  | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|non_existent_counter|6.37e+12|368934881474.19104|        -1| 4294967295|
+--------------------+--------+-------------+---------------+-----------+
```

Assume a fourth section counter specifies a performance counter component that Qsys defines to have three section counters only (the default value).

In Example 3, the test is performed on a hardware design that does not have any other component defined with registers mapped immediately after the registers of the performance counter component. Therefore, there is no impact to other component. Depending on how you configure the component register base addresses in Qsys for a particular hardware design, unpredictable system behavior could occur.

Output From a printf() or perf_print_formatted_output() Call Near the End of main() Might Be Prematurely Truncated

This issue occurs when the Nios II application executes a `BREAK` instruction to transfer profiling data to the development workstation during the `exit()` or `return()` from `main()`.

As a workaround, call `usleep(500000)` before exiting or returning from `main()`. This call creates an adequate delay for you to transmit the I/O to the JTAG UART before `main` returns (or calls `exit()`). If the output is still partially truncated, increase the delay value passed to `usleep()`. Use `#include <unistd.h>` for the `usleep()` function prototype.

Fitting a Performance Counter in a Hardware Design That Consumes Most of a Device's Resources

During development, you can measure the system in a larger device than the size of your device in a deployed system.

Configure a performance counter to have only one section counter to save the most resources.

The Histogram for the gmon.out File Is Missing, Even Though My main() Function Terminates

If you do not define a system timer for the system, the profiler does not call the `nios2_pcsample()` function, and does not generate the histogram for the **gmon.out**. Define a system timer for your system.

Further Reading

For information about the GNU profiler, refer to the Nios II GNU profiler documentation, included with the GCC documentation, available at the Nios II Embedded Design Suite Support.

Because Altera has rewritten the `lib-gprof` library, the information in this application note about data collection deviates from Altera's implementation.

For information about the performance counter, refer to the *Performance Counter Core* chapter in the *Embedded Peripherals IP User Guide*. For information about the high-speed timer, refer to the *Timer Core* chapter in the *Embedded Peripherals IP User Guide*.

Document Revision History

Table 1 shows the revision history for this application note.

Table 1. Document Revision History

Date	Version	Changes
July 2010	3.0	This revision incorporates the following changes: ■ Replaced mentions of SOPC Builder with Qsys. ■ Updated "Obtaining the Hardware Design" on page 1, "Obtaining the Software Examples" on page 1, "Program Counter Trace Information" on page 4, "Tutorial: Using the GNU Profiler" on page 7, "Creating the Profiler Software Example" on page 7, "Creating the GNU Profiler Report" on page 8, "Creating and Running the Profiler Software Example" on page 9"Analyzing the GNU Profiler Report" on page 10"Flat Profile and Call Graph Example" on page 11, "Modifying the Nios II Hardware Design" on page 14, "Creating the Performance Counter Software Example" on page 15, "Running the Performance Counter Software Example" on page 15, and "Performance Counter Example with Nios II SBT for Eclipse" on page 16.
May 2010	2.0	This revision incorporates the following changes: ■ Updated document, software and screen shots for the Nios II SBT for Eclipse ■ Added the Nios II SBT for Eclipse flow ■ Updated examples for the NEEK
July 2008	1.3	This revision incorporates the following changes: ■ Updated document for the Quartus II software and Nios II EDS v8.0. ■ Replaced references to the Nios II IDE with instructions in the Nios II software build flow. ■ General updates for the Quartus II software v8.0.
February 2006	1.2	Updated document for the Quartus II software and Nios II EDS v5.1 SP1.
November 2005	1.1	Updated document for the Quartus II software and Nios II EDS v5.1.
August 2005	1.0	Initial release.