SoC Watch for Android* 1.4

Document Version 1.4
April, 2014

Faycal Benmlih, Jay Chheda, Davin Daniels,
Grant Haab, Robert Knight, Sarath Nandu,
Jeff Olivier, Hrabri Rajic, Suraj Singh,
Gautam Upadhyaya, Darrell Woods

faycal.benmlih@intel.com, jay.chheda@intel.com, davin.daniels@intel.com,
grant.haab@intel.com, robert.knight@intel.com, sarath.nandu.ramachandran.nair@intel.com,
jeffrey.v.olivier@intel.com, hrabri.rajic@intel.com, surajx.singh@intel.com,
gautam.upadhyaya@intel.com, darrell.s.woods@intel.com

Contents

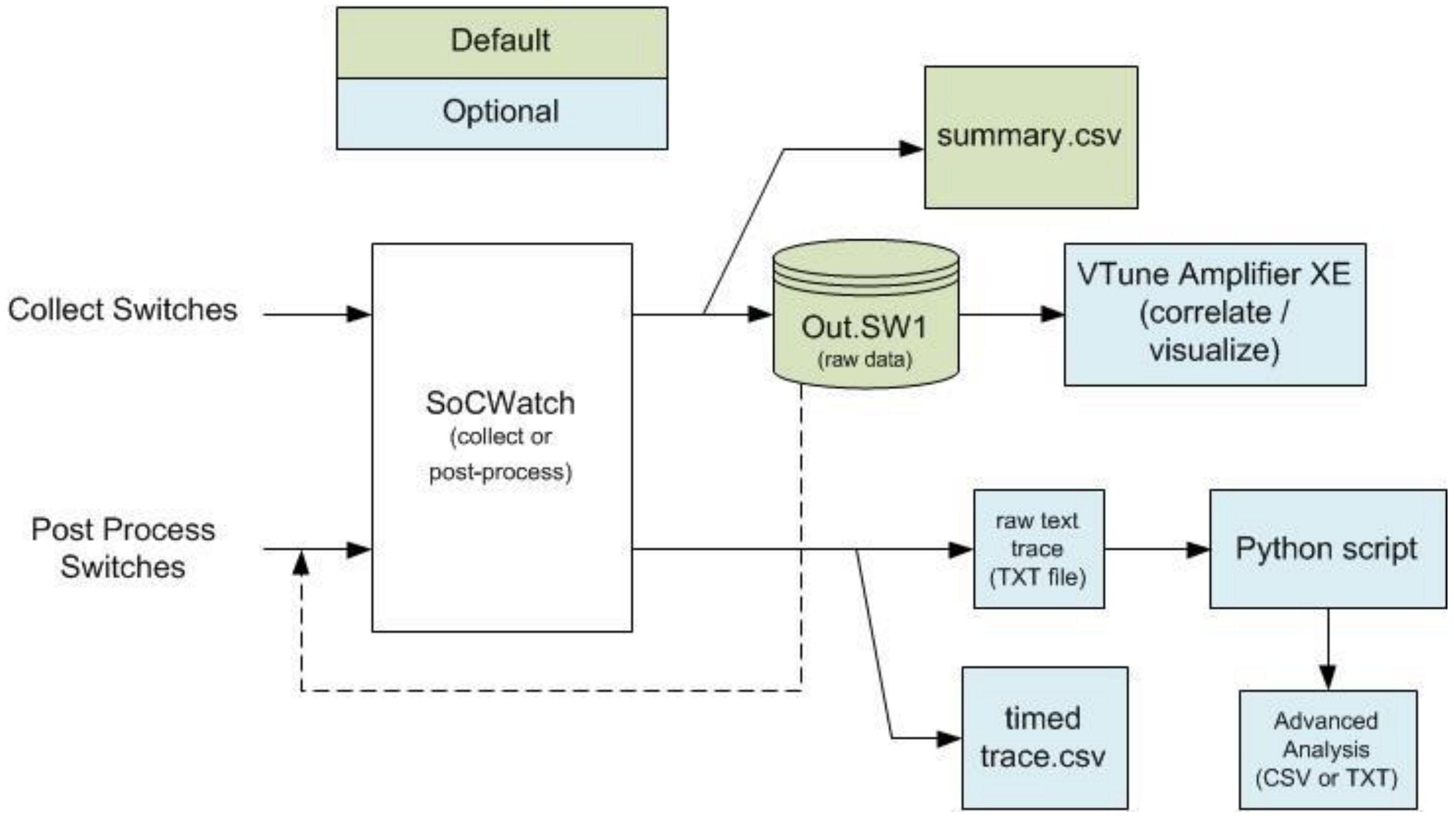
Overview

SoC Watch is a command line tool for monitoring system behaviors related to power consumption on Intel® architecture-based platforms. It monitors power states, frequencies, bus activity, wakeups, Android* wakelocks, and various other metrics that provide insight into the system's energy efficiency.

When running on Android-based systems, SoC Watch also supports monitoring both user and kernel wakelocks. The user wakelock information is available if the associated wakelock patch is applied to the Android* framework.

To collect and analyze power data:

1. Collect data. A summary (CSV) file and raw data (SW1) file are produced by default on the *target* system (which is the system being analyzed).
2. Optionally, import the raw data into the Intel® VTune™ Amplifier tool on the *host* system (which is the system where VTune Amplifier is installed) to correlate and visualize the system behavior over time. Utilize powerful filtering in VTune Amplifier to analyze the results.
3. Optionally, post-process the SW1 file to generate a timed trace (CSV) file that can be opened by Microsoft* Excel* and graph the metrics per time unit.
4. Optionally, post-process the SW1 file to generate a raw trace (TXT) file that can be further analyzed with a provided python script on a host system. The analysis provides additional result data that is not available in any other result.



Requirements

SoC Watch for Android-based systems works on all platforms code named Baytrail, Harris Beach, Merrifield, and Moorefield. See the socwatch.intel.com wiki for detailed system requirements.

Building the Kernel Modules

If the SoC Watch device drivers are not preinstalled in the OS image, you will need to build them.

1. Copy the `socwatch_android.zip` file to the host system used to build the Android* kernel.
2. Extract the contents with the command:
   ```
   > tar xvzf socwatch_android.zip
   ```
   The `socwatch_android` directory is created.
3. `cd` into the `socwatch_android/visa_driver/src` directory.
4. Execute the `build-driver` script with the command:
   ```
   > sh ./build-driver
   ```
5. `cd` into the `socwatch_android/socwatch_driver/lx_kernel` directory.
6. Execute the `build_driver` script with the command:
   ```
   > sh ./build_driver -k <kernel-build-dir>
   ```
   where `<kernel-build-dir>` is replaced with the local Android* lib/modules directory produced while building the Android* kernel. The `-k` switch is optional and is not required if the `DEFAULT_KERNEL_BUILD_DIR` value is properly set in the `build_driver` script.

Installation

The SoC Watch device drivers are now integrated into many of Intel's internal Android* distributions. Note that the socwatch executable files are not integrated into these Android* distributions and need to be installed with the socwatch_android_install script or BAT files. Make sure the host is connected to the target via `adb` before running the install script.

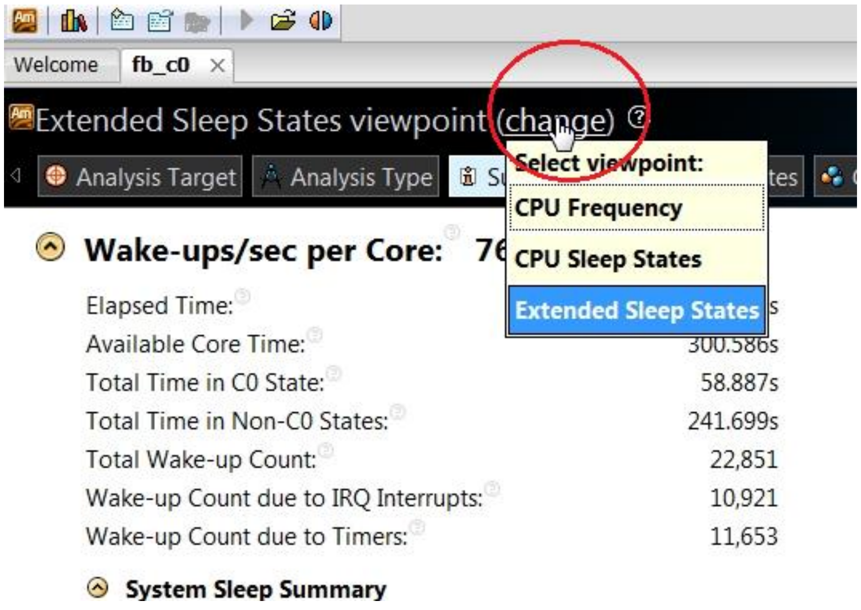
1. After unzipping the SoC Watch package on your host system, run the `socwatch_android_install.sh` script on a Linux host or from a Cygwin window on a Windows host. Run the `socwatch_android_install.bat` file from a Windows host.
   ```
   > socwatch_android_install.sh or socwatch_android_install.bat
   ```
   The script installs the socwatch executables to the `/data/socwatch` directory on the target by default. Use the –d option to select a different install directory and the –s option to define a specific Android* device if multiple devices are connected to the host via adb.
2. Using the `adb` command, start a shell with root privileges.
   ```
   > adb root
   > adb shell
   ```
3. Navigate to the directory containing the drivers.
   ```
   > cd /lib/modules
   ```
4. Load the drivers. Note that the intel_visa driver must be loaded before the socwatch driver.
   ```
   > insmod intel_visa1_1.ko
   ```
   Note: the latest version of the intel_visa driver should be used
   ```
   > insmod socwatch1_4.ko
   ```
5. Confirm the drivers are loaded.
   ```
   > lsmod
   ```
6. When necessary, type `rmmod intel_visa1_1` and `rmmod socwatch1_4` to unload the drivers. The socwatch driver must be unloaded before the intel_visa driver is unloaded.

Quick Start

The following steps assume the SoC Watch drivers and executables are installed. See the *Installation* section above for instructions on how to install SoC Watch.

Use the following steps, targeted for a Microsoft* Windows*-based host, to quickly collect processor C-state and P-state data for 60 seconds on an Android-based system and import it into VTune Amplifier for analysis.

1. On the host system, establish a root adb shell on the target.
   ```
   > adb root
   > adb shell
   ```
2. Load the device drivers. Note that the intel_visa driver must be loaded before the socwatch driver.
   ```
   > insmod /lib/modules/intel_visa1_1.ko
   ```
   Note: the latest version of the intel_visa driver should be used
   ```
   > insmod /lib/modules/socwatch1_4.ko
   ```
3. Confirm the drivers are loaded.
   ```
   > lsmod
   ```
   Confirm the intel_visa and socwatch drivers are included in the list of installed modules.
4. Setup the collection environment. This step assumes the default install directory was used.
   ```
   > cd /data/socwatch
   > . ./setup_socwatch_env.sh
   ```
5. Collect data and generate the `test.csv` and `test.sw1` files in the results directory. This step assumes the `/data/socwatch/result` directory exists.
   ```
   >./socwatch --max-detail –f cpu-cstate –f cpu-pstate –t 60 –o
   ./results/test
   ```
6. Exit the adb shell.
   ```
   > exit
   ```
7. Use adb to pull the result files to the host.
   ```
   > adb pull /data/socwatch/results/test.csv c:\results
   > adb pull /data/socwatch/results/test.sw1 c:\results
   ```
8. Review the summary. Open the `c:\results\test.csv` file with Microsoft* Excel*.
9. Import the data into VTune Amplifier (adjust the paths for Linux):
   Open a command prompt. Note that file write permissions are required for the working directory.
   Specify the full path to the VTune Amplifier amplxe-runss executable and enter:
   ```
   > "<VTune Amplifier installation folder>\bin32\amplxe-runss.exe" –
   import-socwatch-data c:\results\test.sw1 –r c:\results\test
   ```
   This command creates the `c:\results\test` directory. The `test.amplxe` file is created along with additional files and directories. To avoid specifying the whole path, execute the `amplxe-vars.bat` or `amplxe-vars.sh` file in the VTune installation directory.
10. Navigate to the VTune Amplifier `bin32` or `bin64` directory and start the VTune Amplifier GUI.
    ```
    > amplxe-gui.exe
    ```
11. Open the result by selecting: **File -> Open -> Result**... and navigating to the `c:\results\test\test.amplxe` file. Click **Open**.
12. Click on the Viewpoint selection drop-down **(change)** link and select the **Extended Sleep States** viewpoint to view and analyze the data.



Wake-ups/sec per Core: 76

Elapsed Time:	s
Available Core Time:	300.586s
Total Time in C0 State:	58.887s
Total Time in Non-C0 States:	241.699s
Total Wake-up Count:	22,851
Wake-up Count due to IRQ Interrupts:	10,921
Wake-up Count due to Timers:	11,653

To collect additional data, see the *Using SoC Watch* section below. To graph the data collected in Microsoft* Excel*, see the *Graphing Timed Trace Data with Microsoft* Excel** section below. To generate advanced analysis of the data with the `socwatch_advanced_analysis_v1_4.py` python script, see the *Advanced Analysis of Raw Data* section below.

Key Files

`socwatch_android_install.sh`	The Linux install script. It creates a `/data/socwatch` directory on the target system and copies the necessary files there via the `adb push` shell command.
`socwatch_android_install.bat`	The Windows install script. It creates a `/data/socwatch` directory on the target system and copies the necessary files there via the `adb push` shell command.
`visa_driver/src/build-driver`	The build script used to build the intel_visa device driver for Android* systems.
`/lib/modules/intel_visa1_1.ko`	The visa kernel module used to program and read VISA/SoCHaP registers at runtime.
`socwatch_driver/lx_kernel/build_driver`	The build script used to build the SoC Watch device driver for Android* systems.
`/lib/modules/socwatch1_4.ko`	The socwatch kernel module used to collect both hardware and kernel data at runtime.
`setup_socwatch_env.sh`	The script used to setup the socwatch runtime environment.
`socwatch`	The SoC Watch executable built as a native Android* application. Use this file on Android* systems to collect data and generate additional results from a raw SW1 file.
`SOCWatchConfig.txt`	The SoC Watch configuration file. The

	configuration file is read by SoC Watch immediately before each collection. It contains hardware addresses utilized by the device driver during the collection.
`socwatch_advanced_analysis_v1_4.py`	A python script that generates advanced analysis results from the SoC Watch raw text output.
`EULA.txt`	End User License Agreement file.

Using SoC Watch

Caution: *SoC Watch does not collect any data by default. You must specify what data you wish to collect with the –f switch.*

Before starting a collection, source the `setup_socwatch_env.sh` script to set the necessary environment variables.

> **Note:** The first . character followed by a space must be specified in order to set the environment variables for the shell. Failure to include the initial character (period) will cause the `socwatch` execution step to fail.

```
. ./setup_socwatch_env.sh
```

To launch a collection, type

```
./socwatch <general options> <collection options> <post-processing options>
```

General Options

`-c, --config-file <config_file_name>`: Specify the configuration file. The `<config_file_name>` argument can optionally include an absolute or relative path. SoC Watch loads architecture information from a configuration file, for example, MSR address information. If this switch is not specified, SoC Watch loads the default configuration file `SoCWatchConfig.txt`.

`-d, --debug`: Generate debug output from SoC Watch. Used to debug SoC Watch collection and processing issues.

`-h, --help`: Display usage information and exit.

`-v, --version`: Print SoC Watch and device driver information and exit.

Collection Options

`-f, --feature <data_to_collect>`: Specify the data to be collected. See the *Available Features* section below. Each `data_to_collect` needs to be prefaced with a `-f` or `--feature` switch. For example, `-f cpu-cstate` will cause SoC Watch to collect processor C-state data and `-f cpu-cstate -f cpu-pstate` will cause SoC Watch to collect both processor C-state and processor P-state data. The `-m, --max-detail` switch determines what collection mechanism is used. See below.

`-m, --max-detail`: Specify the level of detail to collect. This switch applies to all data specified to be collected by the `-f` switch. Internally, this switch setting guides the tool's choice of a collection mechanism and also controls overhead. In general, if `-m` is specified, the tool will collect the maximum detail possible and may also cause the highest overhead. In general, if `-m` is not specified, the tool will collect a minimum amount of detail and may also cause the minimum overhead.

SoC Watch uses different methods to collect data to provide the maximum detail or minimum overhead. Data may be traced, polled, or snapshot to give you the most accurate result or an acceptable result with minimum system perturbation. Traced data is read when the metric itself changes providing the most accurate result. Polled data is read at regular intervals. Note that polled data may not be 100% accurate as metric changes that take place between polling intervals will not be measured. Snapshot data is read at the beginning and end of the collection and the difference is provided in the result. Snapshot data provides the average result, minimum overhead, and the least amount of detail. The algorithm used to determine the collection method for each data type is as follows:

```
If -m is specified;
  if the data can be traced, trace it;
  else if the data can be polled, poll it;
  else snapshot it.
If -m is not specified;
  if the data can be snapshot, snapshot it;
  else if the data can be traced, trace it;
  else poll it.
```

`-n, --interval <milliseconds>`: Specify the polling interval used when data is polled. The default polling interval is 100ms. The minimum polling interval is 1ms.

`-o, --output <file_name>`: Specify an output file name (optionally including path). Collected data (if any) is written to the file `file_name.csv` and `file_name.sw1`. If the `-o` switch is not specified, data is saved to the `SoCWatchOutput.csv` and `SoCWatchOutput.sw1` files in the current directory. To create additional files, use the `-r` switch described below.

`-p, --program <application><parameters>`: Specify the application (including path to the executable) and its parameters to be launched by SoC Watch at the same time a collection is started.

Note: The –p switch MUST be the last SoC Watch switch used on the command line.

`-r, --result <result_type>`: Specify the type of results to be generated. Two types are possible: `int` and `raw`. The `int` option generates a timed trace CSV file for each data type collected (specified by the `-f` switch). This switch must be used during a collection. If the `-r raw` switch is used in conjunction with the `-m` switch, a raw trace TXT file will be generated that can be further analyzed with a python script. Each result file to be generated must be prefaced with the `-r` switch. Only traced or polled data can generate an `int` or `raw` result.

`-s, --startdelay <seconds>`: Specify the collection delay in seconds. Data capture starts `seconds` after the SoC Watch command is executed.

`-t, --time <seconds>`: Specify the duration of the collection, in seconds. SoC Watch stops a collection when any of the following three events occur:

a. the duration specified by the `-t` switch has elapsed

b. SoC Watch receives a `SIGINT` (CTRL-C)
c. the application launched by SoC Watch exits

Available Features

The available data types that SoC Watch can collect and their possible collection methods are listed below. Detailed descriptions of the different kinds of data are provided in the *Data Type Descriptions* section below.

cpu-cstate

<u>Collection Method(s):</u> trace, snapshot
Collect IA core, module, and package C-state data using hardware C-state residency MSR data and the wakeups that cause IA cores to exit a C-state. Note that wakeups are only collected if the –m switch is specified.

cpu-pstate

<u>Collection Method(s):</u> trace
Collect IA core P-state (frequency) data using hardware MSR data.

s0i-state, acpi-sstate

<u>Collection Method(s):</u> snapshot, trace (respectively)
If the `s0i-state` or `acpi-sstate` feature is specified, both S0ix and ACPI Suspend-To-RAM (S3) data will be collected concurrently. S3 data is traced using kernel hooks to trace the system's S3 behavior. S0ix data is snapshot at the start of the collection, at the end of the collection, and during S0->S3 and S3->S0 transitions. On the Merrifield or Moorefield platforms, SoC Watch issues IPC commands to the SCU to read S0ix counters. On the Baytrail platform, SoC Watch reads counters in the PMC block's PCI address.

nc-dstate

<u>Collection Method(s):</u> poll
Collect North Complex component D0ix data via PCI reads.

sc-dstate

<u>Collection Method(s):</u> snapshot
Collect South Complex component D0ix data on platforms code named Merrifield or Moorefield only. SoC Watch issues IPC commands to the SCU to read D0ix counters.

wakelock

<u>Collection Method(s):</u> trace
Collect user and kernel wakelock data using kernel tracepoints and data gathered from the aplog.

core-temp

<u>Collection Method(s):</u> poll
Collect IA core temperature data via MSR reads.

pmic-temp

<u>Collection Method(s):</u> poll
Collect MSIC/PMIC temperature data via sysfs reads.

skin-temp

<u>Collection Method(s):</u> poll
Collect device skin temperature data via sysfs reads.

soc-temp
Collection Method(s): poll
Collect SoC temperature data via PCI reads.

gfx-pstate
Collection Method(s): poll
Collect GPU P-state (frequency) data via PCI reads.

gfx-cstate
Collection Method(s): poll
Collect GPU C-state data via PCI reads. This data type can only be collected on platforms code named Baytrail.

ddr-bw
Collection Method(s): poll
Measure total DDR bandwidth in MB/sec using the system's VISA/SoCHaP functionality. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

cpu-ddr-bw
Collection Method(s): poll
Measure the IA Core to DDR bandwidth in MB/sec using the system's VISA/SoCHaP functionality. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

io-bw
Collection Method(s): poll
Measure the bandwidth between the North Cluster (NC) and South Cluster (SC) in MB/sec using the system's VISA/SoCHaP functionality. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

disp-ddr-bw
Collection Method(s): poll
Measure the display controller to DDR bandwidth in MB/sec using the system's VISA/SoCHaP functionality. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

gfx-ddr-bw
Collection Method(s): poll
Measure the graphics component to DDR bandwidth in MB/sec using the system's VISA/SoCHaP functionality. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

isp-ddr-bw
Collection Method(s): poll
Measure the ISP (camera image processor) to DDR bandwidth in MB/sec using the system's VISA/SoCHaP functionality. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

all-approx-bw
Collection Method(s): poll
Estimate multiple SoC bandwidths concurrently. SoC transactions are counted and multiplied by 64 to estimate multiple SoC bandwidths concurrently. The estimates will be equal to or higher than the actual SoC bandwidths. If the transactions executed during a collection include partial or 32 byte transfers, the estimate will be high.

On systems code named Baytrail, this feature provides the following estimated bandwidths;

Module 0 and **Module 1**: the bandwidth from CPU Module 0 and Module 1 to the DDR
GFX: the bandwidth from the graphics component to the DDR
Display: the bandwidth from the display controller to the DDR
ISP: the bandwidth from the camera image processor to the DDR
VED: the bandwidth from the video encode and decode components to the DDR
IO: the bandwidth between the north and south clusters

On systems code named Merrifield or Moorefield, this feature provides the following estimated bandwidths;
Module 0: the bandwidth from CPU Module 0 to the DDR
Display: the bandwidth from the display controller to the DDR
GFX: the bandwidth from the graphics component to the DDR
ISP: the bandwidth from the camera image processor to the DDR
IO: the bandwidth between the north and south clusters

dram-srr
dram-srr-ch0
dram-srr-ch1
Collection Method(s): poll
Measure the residency of DRAM while it is in self refresh mode using the system's VISA/SoCHaP functionality. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

The ch0 and ch1 versions of this feature are intended to be used on systems code named Baytrail.

Combination Features

The following features are combinations of the previously described features. These features can be used to simplify command lines used to collect multiple data types concurrently. For example, the `-f cpu` switch can replace the `-f cpu-cstate -f cpu-pstate` switches in a command line.

If a combination feature includes a feature not supported on the target platform, that feature will be ignored. For example, if the `device` switch is utilized on a system code named Baytrail, the `sc-dstate` feature, normally included in the `device` feature, will be ignored.

cpu

A combination of the `cpu-cstate` and `cpu-pstate` features.

device

A combination of the `nc-dstate` and `sc-dstate` features.

gfx

A combination of the `gfx-cstate` and `gfx-pstate` features.

sys

A combination of the `cpu-cstate`, `cpu-pstate`, `s0i-state`, `acpi-sstate`, `nc-dstate`, `sc-dstate`, `core-temp`, `soc-temp`, `skin-temp`, `pmic-temp`, `gfx-cstate`, `gfx-pstate`, and `ddr-bw` features.

temp

A combination of the `core-temp`, `soc-temp`, `skin-temp`, and `pmic-temp` features.

Post-Processing Options

`-i, --input <file_name>`: Specify an input SW1 file to post-process. Use this switch in combination with the `-r raw` switch to generate a `file_name.txt` raw trace result.

`-r, --result <result_type>`: Specify the type of results to be generated. One type is possible: `raw`. The `raw` option generates a raw trace TXT file that can be further analyzed with the provided `socwatch_advanced_analysis_v1.4.py` python script. In order to generate a TXT file with the `-r raw` switch, the SW1 file must be generated with the `-m` switch. Only traced or polled data can generate a `raw` result.

The `-o` switch can be used to define the output name of the TXT file generated with the `-i -r raw` switch combination. If the `-o` switch is not specified, the TXT file will be named `SoCWatchOutput.txt` and placed in the working directory.

Examples

```
./socwatch -f cpu-cstate -t 10
```

Collect processor C-state information with minimum overhead and detail – snapshot C-state data for 10 seconds. Generate the `SoCWatchOutput.csv` summary result file and the `SoCWatchOutput.sw1` file that can be post processed or imported into VTune Amplifier.

```
./socwatch -max-detail -f cpu-cstate -t 10
```

Collect processor C-state information with maximum detail – trace C-state data for 10 seconds. Generate the `SoCWatchOutput.csv` summary result file and the `SoCWatchOutput.sw1` file that can be post-processed or imported into VTune Amplifier.

```
./socwatch -i SoCWatchOutput.sw1 -r raw
```

Use the `SoCWatchOutput.sw1` file collected earlier to generate the `SoCWatchOutput.txt` raw trace file. The `SoCWatchOutput.txt` can be further analyzed with the `socwatch_advanced_analysis_v1.4.py` python script. See the *Advanced Analysis of Raw Data* section below.

```
./socwatch -f cpu-cstate -f cpu-pstate -f core-temp -f wakelock -t 120
```

Collect processor C-state, processor P-state, core temperature, and wakelock information with minimum overhead and detail for 120 seconds. Polled data, `core-temp` data in this case, will be read at a 100ms (default) interval. Generate the `SoCWatchOutput.csv` summary result file and the `SoCWatchOutput.sw1` file that can be post processed or imported into VTune Amplifier.

```
./socwatch -f cpu-cstate -f core-temp -n 1000 -t 30 -o ./result/test1
```

Collect processor C-state and core temperature information with minimum overhead and detail for 30 seconds. Polled data will be read at a 1000ms (1 second) interval. Generate the `result/test1.csv` summary result file and the `result/test1.sw1` file that can be post processed or imported into VTune Amplifier.

SoC Watch Output Files

SoC Watch generates two files by default after every collection including a summary CSV file and a raw data SW1 file. SoC Watch can optionally generate timed trace CSV files for each feature (data type) collected and a raw trace TXT file that can be further analyzed with an included python script.

Summary CSV File

The summary CSV file provides an overall summary of the data collected for each feature (data type) specified with the `–f` switch.

Raw Data SW1 File

The raw data SW1 file captures the raw results of the collection in a binary format. The SW1 file can be imported and visualized with VTune Amplifier.

Timed Trace CSV Files

A timed trace CSV file for each feature (data type) collected can be optionally generated with the `–r int` switch. For example, a timed trace CSV file can be generated that describes the processor C-state results on a per second basis. The resulting file can be imported into Microsoft* Excel* to be graphed.

Raw Trace Text File

A raw trace TXT file can be optionally generated with the `–r raw` switch. This file can be input to a python script provided in the SoC Watch package. The python script analyzes the raw results to generate advanced analysis of the collection results.

Data Type Descriptions

The following section describes the different data types that can be collected by SoC Watch. Each subsection's title corresponds to the `–f` switch used to collect a particular data type.

cpu-cstate

The `–f cpu-cstate` switch measures IA core C-states. C-states are core power states used to minimize the core's power consumption when the core is not actively executing instructions. C-states range from C0 to Cn (for example, C6). C0 is the power state where the core is actively executing instructions. Deeper C-states (for example, C1, C2, and others) consume less power than C0 with Cn consuming the least amount of power. Each C-state deeper than C0 has a different enter and exit latency with the deeper C-states taking longer to enter and exit (return to C0).

The operating system requests specific C-states using the `mwait` instruction. The processor may accept the request, select a different C-state based on its own status, or abort the C-state. See the Intel 64 and IA-32 Architectures Software Developer's Manual at http://download.intel.com/products/processor/manual/253669.pdf for more information.

cpu-pstate

The `–f cpu-pstate` switch measures IA core P-states. P-states represent voltage-frequency control states defined as performance states in the industry standard Advanced Configuration and Power Interface (ACPI) specification. See http://www.acpi.info for more details. In voltage-frequency control, the voltage and clocks that drive circuits are increased or decreased in response to a workload. Also known as Dynamic Voltage and Scaling Frequency (DVFS), voltage-frequency control leverages the equation $P=a \times C_{eff} \times V^2 \times f$ that describes the power consumption of a CMOS circuit. P0 represents turbo mode where the processor can operate at a frequency higher than its nominal frequency depending on the state of the other cores, the current thermal headroom, and so on. P0 consumes the most power. P1 is the processor's nominal frequency (advertised frequency), P2 is slower than P1, and so on. Pn consumes the least power.

The operating system requests specific P-states based on the current workload. The processor may accept or reject the request and set the P-state based on its own state. Turbo mode is completely controlled by the hardware.

> **Note**: Operating the processor at the maximum P-state to finish the workload as quickly as possible and enter a deep C-state (known as *Race to Halt*) is often the most efficient power management strategy.

s0i-state, acpi-sstate

The `-f s0i-state` or `-f acpi-state` switch measures the residency in the Intel SoC idle standby power states and the system's residency in the ACPI Suspend-To-RAM (S3) state. The S0ix states shut off parts of the SoC when they are not in use. The S0ix states are triggered when specific conditions within the SoC have been achieved – for example, certain SoC components are in low power states. See http://landley.net/kdocs/ols/2012/ols2012-mansoor.pdf for a full description of the S0ix power states. The SoC consumes the least amount of power in the deepest (for example, S0i3) state.

In the Suspend-To-RAM state, the Linux kernel powers down many of the systems' components while maintaining the system's state in its main memory. The system consumes the least amount of power possible while in the Suspend-To-RAM state. Note that any wakelock currently held will prevent the system from entering the Suspend-To-RAM state. See the description of Suspend-To-RAM at http://www.linuxsymposium.org/archives/OLS/Reprints-2008/brown-reprint.pdf.

nc-dstate

The `-f nc-dstate` switch polls the SoC's component power states for the north cluster components *while the SoC is operating in the S0i0 state*. D0ix power states ranges from D0i0 to D0i3, where D0i0 is fully powered on and D0i3 is primarily powered off.

The SoC is organized into a north and south cluster where the compute intensive components (for example, video decode, image processing, and others) are located in the north cluster. The south cluster contains I/O, audio, system management, and other components. SoC components should be in the D0i3 state when not in use.

sc-dstate

The `-f sc-dstate` switch measures the SoC's component power state residencies for the south cluster components. D0ix power states range from D0i0 to D0i3, where D0i0 is fully powered on and D0i3 is primarily powered off.

> **Note:** The `-f sc-dstate` switch only works on platforms code named Merrifield or Moorefield. See http://landley.net/kdocs/ols/2012/ols2012-mansoor.pdf for a full description of the D0ix power states.

wakelock

The `-f wakelock` switch traces both user and kernel wakelocks used during the collection. Wakelocks are an Android* concept that allows both application and kernel software to prevent the system from entering the ACPI Suspend-to-RAM (S3) state. Application and kernel software hold wakelocks to insure that the system does not enter S3 during a critical operation. Over-use use of wakelocks (for example, holding a wakelock too long or never releasing a wakelock) can cause the system to consume too much power. See http://dl.acm.org/citation.cfm?id=2307661 for a good description of power consumption bugs caused by incorrect use of wakelocks.

core-temp

The `-f core-temp` switch measures the temperature of each core during the collection. Understanding the core's temperature is important as the core will throttle when it gets too hot. See the Intel 64 and IA-32 Architectures Software Developer's Manual at http://download.intel.com/products/processor/manual/253669.pdf for more information.

pmic-temp

The `-f pmic-temp` switch measures the system's Power Management Integrated Circuit's temperature during the collection. Note that the PMIC is also called the Mixed Signal Integrated Circuit (MSIC).

skin-temp

The `-f skin-temp` switch measures the system's skin temperature during the collection. The temperatures are read from the sysfs.

soc-temp

The `-f soc-temp` switch measures the system's SoC temperature during the collection.

gfx-pstate

The `-f gfx-pstate` switch measures the system's GPU frequency during the collection. The system can vary the GPU's P-state during the collection in response to the graphics workload in an effort to minimize power consumption.

gfx-cstate

The `-f gfx-cstate` switch measures the system's GPU power state residency during the collection. GPU power states range from Render C0 to C6 and Media C0 to C6. The C0 states indicate the GPU is fully powered on and operational while the C6 states indicate the GPU has been placed in the deepest power saving state available. Note that this data type can only be collected on platforms code named Baytrail.

ddr-bw

The `-f ddr-bw` switch measures the system's total DDR bandwidth in MB/sec during the collection. The total DDR bandwidth includes all 32 byte and 64 byte read and write accesses to the memory from any component on the SoC including the cores, display, image processing, graphics, and south cluster components. Total DDR bandwidth activity indicates that the SoC is actively executing a workload and is a clue when trying to understand the system's behavior when diagnosing system power consumption issues. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

cpu-ddr-bw

The `-f cpu-ddr-bw` switch measures the system's CPU to DDR bandwidth in MB/sec during the collection. The CPU to DDR bandwidth includes all 32 byte and 64 byte read and write accesses to the memory from the CPU components in the north cluster. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

io-bw

The `-f io-bw` switch measures the system's bandwidth between the north and south cluster in MB/sec during the collection. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

disp-ddr-bw

The `–f disp-ddr-bw` switch measures the system's display controller to DDR bandwidth in MB/sec during the collection. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

gfx-ddr-bw

The `–f gfx-ddr-bw` switch measures the system's GPU to DDR bandwidth in MB/sec during the collection. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

isp-ddr-bw

The `–f isp-ddr-bw` switch measures the system's image processor to DDR bandwidth in MB/sec during the collection. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time.

dram-srr

dram-srr-ch0
dram-srr-ch1

The `–f dram-srr`, `-f dram-srr-ch0`, and `–f dram-srr-ch1` switches measure the systems DRAM self refresh residency during the collection. The systems DRAM will enter a low power self refresh mode when it is not actively being utilized. DRAM self refresh residency should be high during idle workloads. Note that only one bandwidth or DRAM Self Refresh data type can be specified at a time. The ch0 and ch1 versions of the switch are intended to be used on systems code named Baytrail only.

Using VTune Amplifier to Analyze SoC Watch Data

Overview

You can analyze SoC Watch data graphically using the VTune Amplifier GUI. The GUI provides a dynamic timeline view for interacting with SoC Watch data and provides powerful filtering of data for in-depth analysis of a platform's power management behavior. VTune Amplifier or later is recommended to analyze data collected with SoC Watch v1.4.

Importing Data

To import data using VTune Amplifier, perform the following steps:

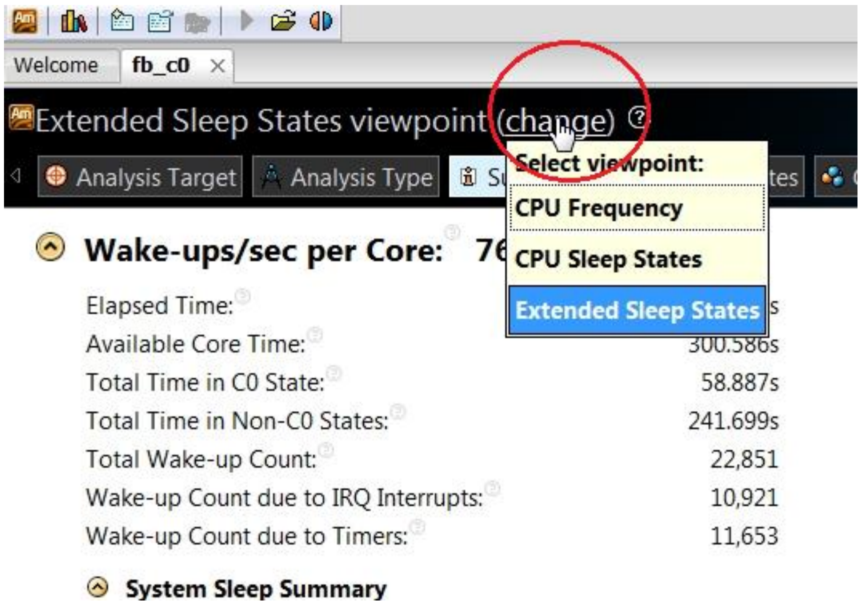
1. Collect SoC Watch raw data (an SW1 file) as you normally would on your target system. You may specify any output file name, but these instructions are written assuming you use the default output file name `SoCWatchOutput.sw1`.
2. Copy the raw data file back to the host development system:
   ```
   > adb pull /path/SoCWatchOutput.sw1
   ```
3. Import the data into a VTune Amplifier:
   ```
   amplxe-runss –import-socwatch-data <path/SoCWatchOutput.sw1> <-r
   result_dir_name>
   ```
   This creates a directory called `result_dir_name` with a `result_dir_name.amplxe` file. If you do not supply the (optional) result directory name, VTune Amplifier creates a directory called `r###` with an `r###.amplxe` file in the current directory.

   Note: File write permissions are required for the working directory used during the import command. To ensure you have file write permissions, open the command prompt with administrator privileges.

4. Launch the VTune Amplifier GUI:
   - Linux: `/path-to-vtune/bin32(bin64)/amplxe-gui`
   - Windows: Locate the VTune Amplifier in the Start menu.
5. Click `File->Open->Result…` and then browse to your new result directory and select the `.amplxe` file.

Using the VTune Amplifier GUI

Once the VTune Amplifier GUI is loaded, change to the Extended Sleep States viewpoint.



Wake-ups/sec per Core: 76

Elapsed Time:	s
Available Core Time:	300.586s
Total Time in C0 State:	58.887s
Total Time in Non-C0 States:	241.699s
Total Wake-up Count:	22,851
Wake-up Count due to IRQ Interrupts:	10,921
Wake-up Count due to Timers:	11,653

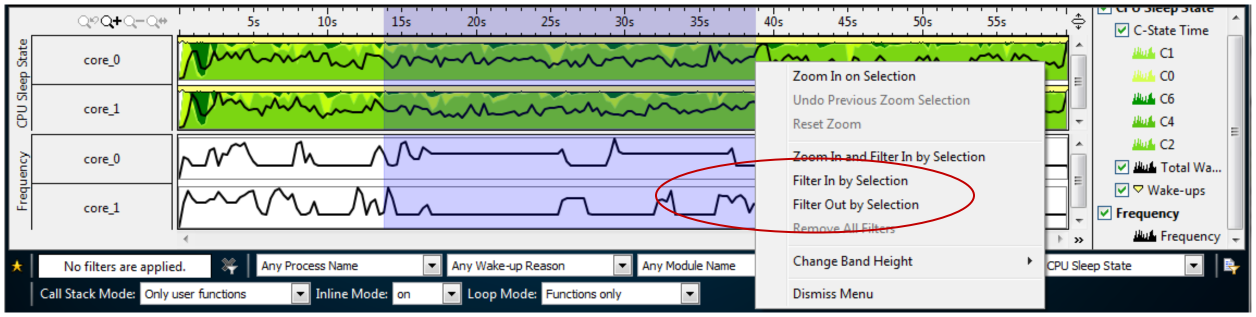
This viewpoint has a tab for each type of SoC Watch data collected. All of the tabs have the same basic layout consisting of a timeline, grid and filter area. These areas provide various mechanisms for manipulating and viewing the data.

Filtering

Each of the areas enables you to filter data to help focus the analysis on specific areas of the collection. Note that a filter from one part of the GUI affects the data shown in other parts of the GUI. This includes data not only within a single tab, but also includes data shown in other tabs.

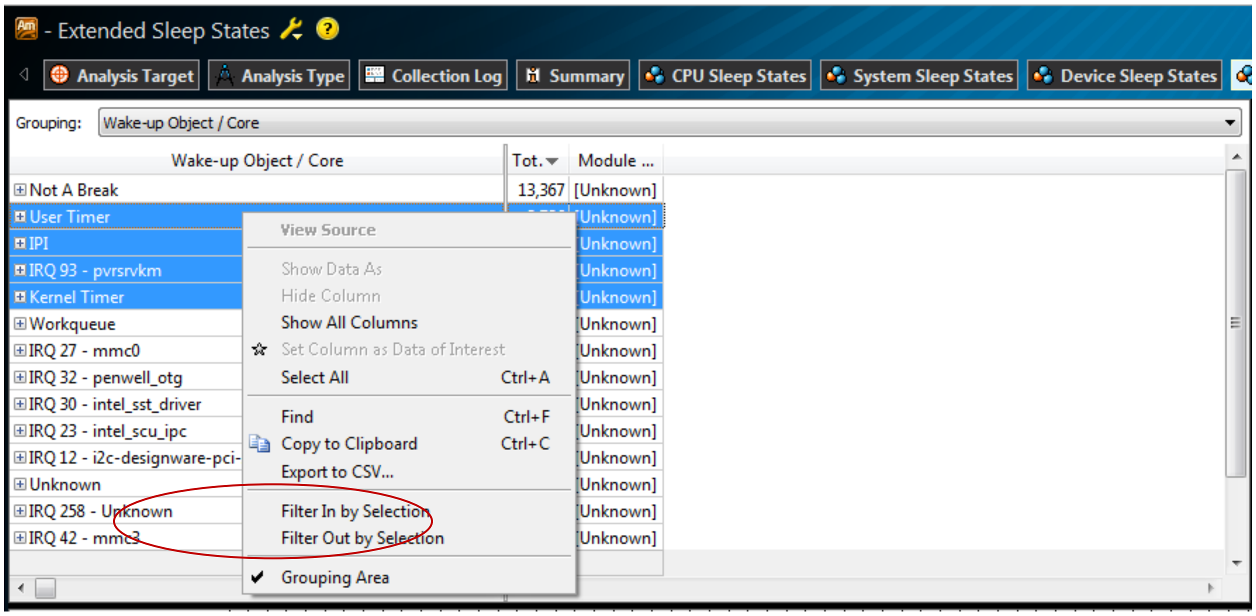
The most common filtering is selecting a region of time to focus on. To do this, use the timeline by selecting a region of time and selecting a filter action. This restricts the data in the grid to only the data that was collected during the selected period of time.

A common choice is the `Zoom In and Filter In by Selection` filter. This filter zooms the timeline and adjusts the grid to include only the data shown in the timeline (for example, the grid only includes the C-state residency or wakeups shown in the zoomed timeline).



Call Stack Mode: Only user functions | Inline Mode: on | Loop Mode: Functions only

It is also possible to filter data using the grid by selecting rows of data, right-clicking to bring up the menu and selecting the filter action.

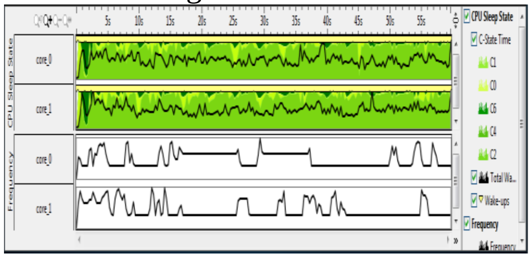


Grouping: Wake-up Object / Core

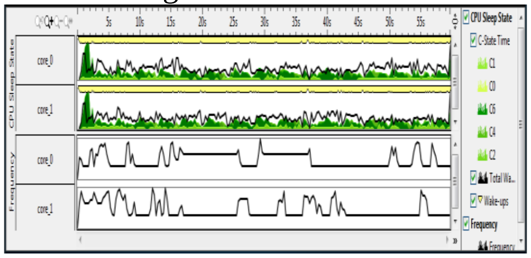
Wake-up Object / Core	Tot.	Module ...
Not A Break	13,367	[Unknown]
User Timer		[Unknown]
IPI		[Unknown]
IRQ 93 - pvrsrvkm		[Unknown]
Kernel Timer		[Unknown]
Workqueue		[Unknown]
IRQ 27 - mmc0		[Unknown]
IRQ 32 - penwell_otg		[Unknown]
IRQ 30 - intel_sst_driver		[Unknown]
IRQ 23 - intel_scu_ipc		[Unknown]
IRQ 12 - i2c-designware-pci-		[Unknown]
Unknown		[Unknown]
IRQ 258 - Unknown		[Unknown]
IRQ 42 - mmc3		[Unknown]

This filter then updates the timeline to show only the data associated with the selected rows in the grid. For example, this is useful when looking at data in the CPU Wake-up tab. By filtering on the selected rows, you can see the behavior of just those wakeup sources in the timeline.

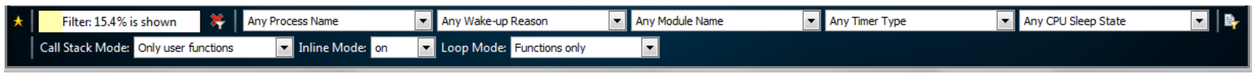
Before filtering



After filtering



The filter area behaves similar to the grid filter in action and is useful for quickly doing the most common and simple filtering actions. The filter area also has the button used to remove all the filters.



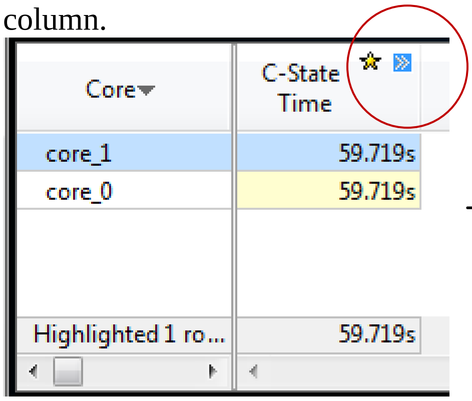
Filter: 15.4% is shown | Any Process Name | Any Wake-up Reason | Any Module Name | Any Timer Type | Any CPU Sleep State

Call Stack Mode: Only user functions | Inline Mode: on | Loop Mode: Functions only

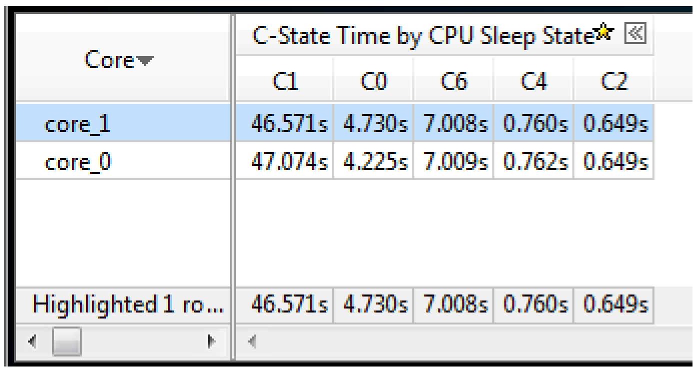
GUI Tips and Tricks

Expanding Grid Columns

The grid has an ability to expand certain columns of data to obtain more information. If a column is expandable, you see a clickable >> symbol in the header of the column. Clicking on this symbol expands the column.

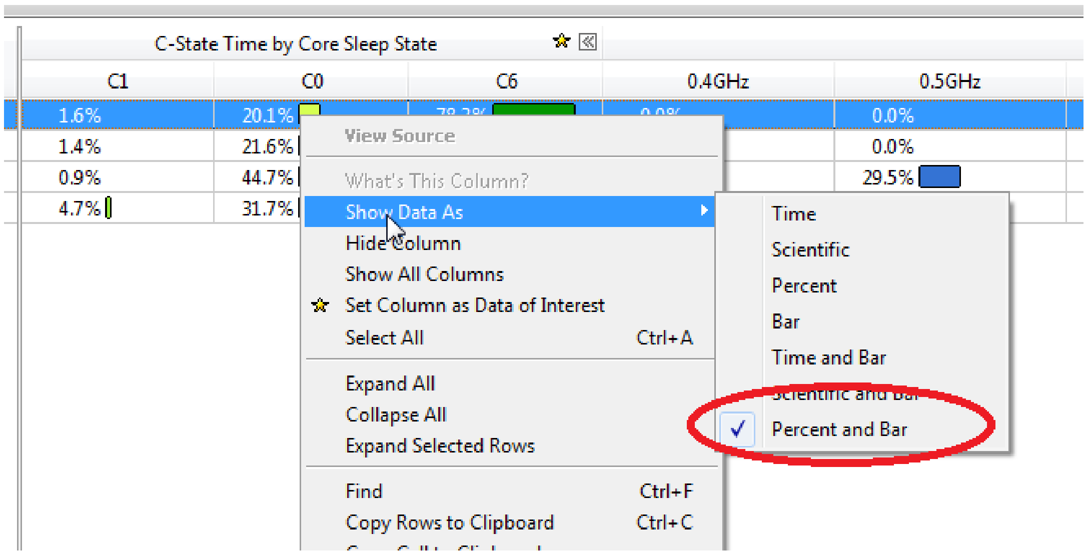


Core	C-State Time
core_1	59.719s
core_0	59.719s
Highlighted 1 ro...	59.719s



Core	C-State Time by CPU Sleep State				
	C1	C0	C6	C4	C2
core_1	46.571s	4.730s	7.008s	0.760s	0.649s
core_0	47.074s	4.225s	7.009s	0.762s	0.649s
Highlighted 1 ro...	46.571s	4.730s	7.008s	0.760s	0.649s

Grid data can also be shown as raw values (for example, time in a specific C-state), percentages, or other representations. Right click on a value in the grid and select **Show Data As** and then select the desired representation.

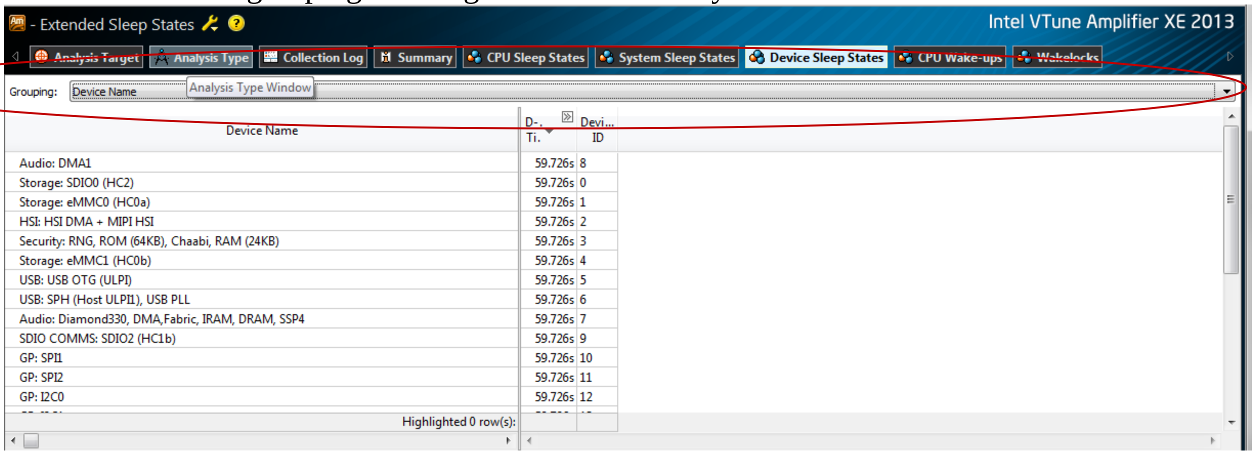


C-State Time by Core Sleep State

C1	C0	C6	0.4GHz	0.5GHz
1.6%	20.1%	78.2%	0.0%	0.0%
1.4%	21.6%			0.0%
0.9%	44.7%			29.5%
4.7%	31.7%			

Changing Grid Groupings

Certain tabs have grids that can show the data using multiple row hierarchies. Use the **Grouping** drop-down box and select the desired grouping to change the row hierarchy.



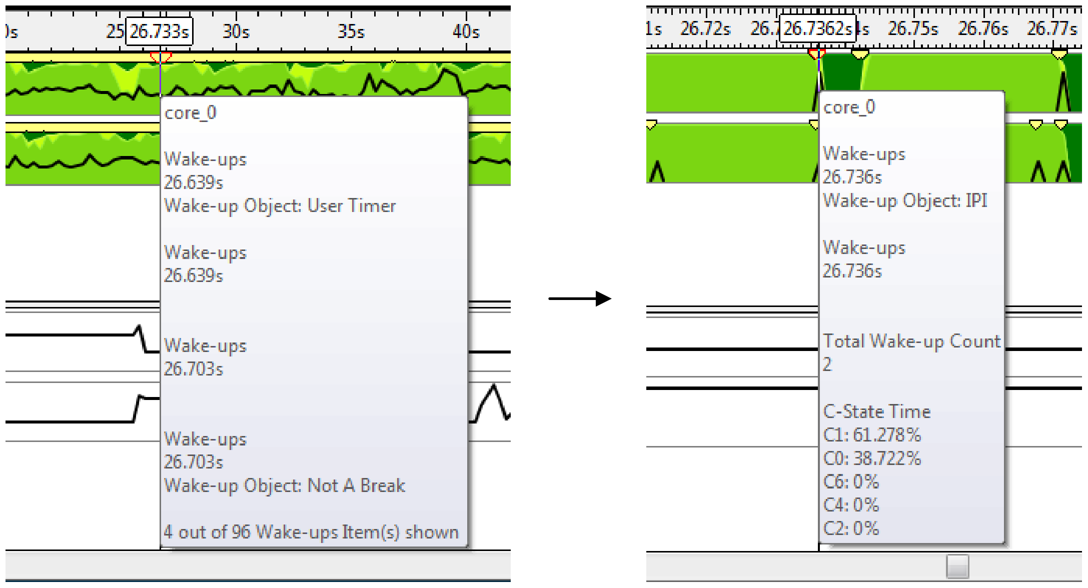
Grouping: Device Name

Device Name	D-. Ti.	Devi... ID
Audio: DMA1	59.726s	8
Storage: SDIO0 (HC2)	59.726s	0
Storage: eMMC0 (HC0a)	59.726s	1
HSI: HSI DMA + MIPI HSI	59.726s	2
Security: RNG, ROM (64KB), Chaabi, RAM (24KB)	59.726s	3
Storage: eMMC1 (HC0b)	59.726s	4
USB: USB OTG (ULPI)	59.726s	5
USB: SPH (Host ULPI1), USB PLL	59.726s	6
Audio: Diamond330, DMA,Fabric, IRAM, DRAM, SSP4	59.726s	7
SDIO COMMS: SDIO2 (HC1b)	59.726s	9
GP: SPI1	59.726s	10
GP: SPI2	59.726s	11
GP: I2C0	59.726s	12

Highlighted 0 row(s):

Timeline Zooming

Using the zoom functionality in the timeline makes it possible to do finer grain filtering and to examine individual discrete events. For example, when zoomed out, this tooltip says that there are 96 wakeups at a given point. By zooming in, it is possible to select each of these individual wakeups.



core_0

Wake-ups
26.639s
Wake-up Object: User Timer

Wake-ups
26.639s

Wake-ups
26.703s

Wake-ups
26.703s
Wake-up Object: Not A Break

4 out of 96 Wake-ups Item(s) shown

core_0

Wake-ups
26.736s
Wake-up Object: IPI

Wake-ups
26.736s

Total Wake-up Count
2

C-State Time
C1: 61.278%
C0: 38.722%
C6: 0%
C4: 0%
C2: 0%

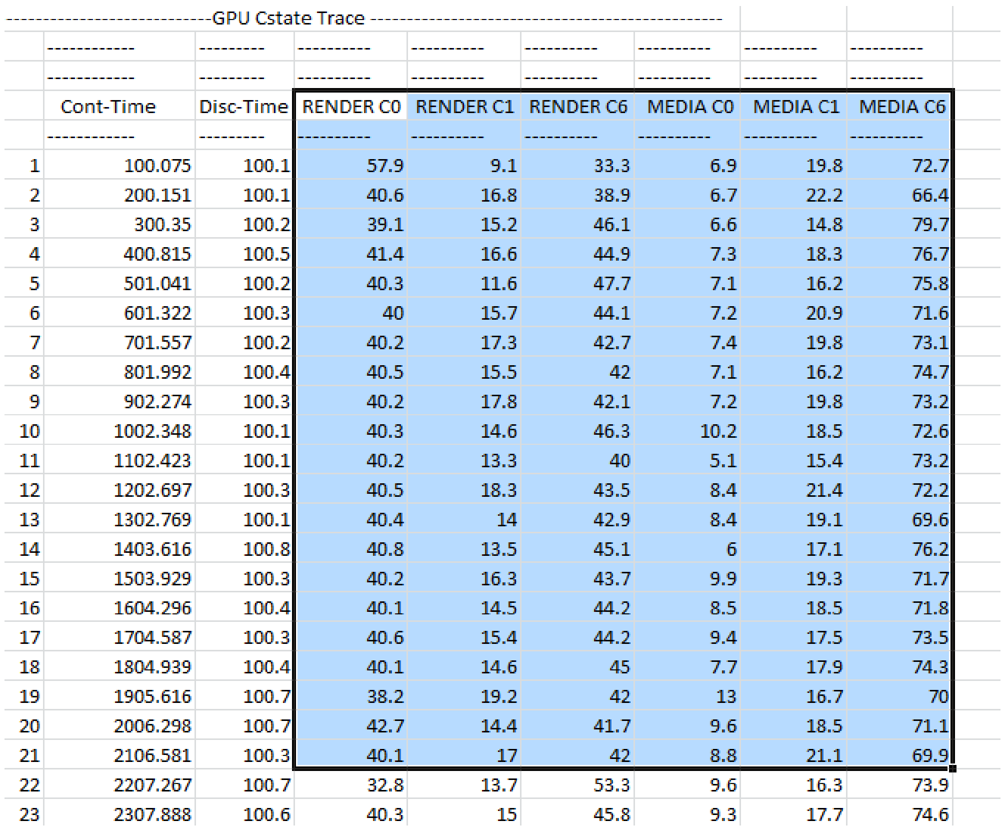
Note: The last sample shown in the VTune Amplifier timeline for polled data types (for example, temperatures, bandwidths, gfx-pstate and nc-dstate) is extrapolated from the previous sample. This is done to ensure that the timeline duration is the same for all metrics.

Graphing Timed Trace Data with Microsoft* Excel*

Per metric, timed trace files (CSV files) generated with the `–r int` switch can be opened and graphed with Microsoft* Excel*.

Follow these steps below to open and graph data in Microsoft* Excel*. The `Trace_gfx-cstate-s0cwatch_20.csv` file is used in this example.

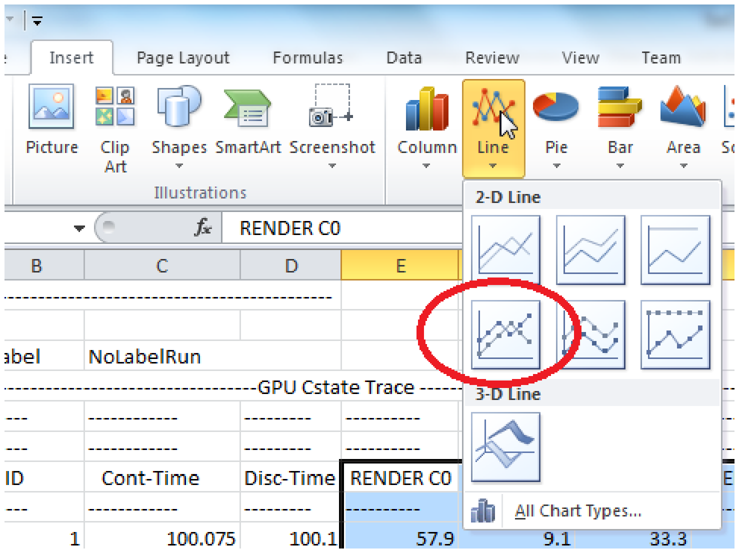
1. Open the CSV file by double clicking on it. Select the data to be graphed.



GPU Cstate Trace

	Cont-Time	Disc-Time	RENDER C0	RENDER C1	RENDER C6	MEDIA C0	MEDIA C1	MEDIA C6
	------------	---------	----------	----------	----------	----------	----------	----------
1	100.075	100.1	57.9	9.1	33.3	6.9	19.8	72.7
2	200.151	100.1	40.6	16.8	38.9	6.7	22.2	66.4
3	300.35	100.2	39.1	15.2	46.1	6.6	14.8	79.7
4	400.815	100.5	41.4	16.6	44.9	7.3	18.3	76.7
5	501.041	100.2	40.3	11.6	47.7	7.1	16.2	75.8
6	601.322	100.3	40	15.7	44.1	7.2	20.9	71.6
7	701.557	100.2	40.2	17.3	42.7	7.4	19.8	73.1
8	801.992	100.4	40.5	15.5	42	7.1	16.2	74.7
9	902.274	100.3	40.2	17.8	42.1	7.2	19.8	73.2
10	1002.348	100.1	40.3	14.6	46.3	10.2	18.5	72.6
11	1102.423	100.1	40.2	13.3	40	5.1	15.4	73.2
12	1202.697	100.3	40.5	18.3	43.5	8.4	21.4	72.2
13	1302.769	100.1	40.4	14	42.9	8.4	19.1	69.6
14	1403.616	100.8	40.8	13.5	45.1	6	17.1	76.2
15	1503.929	100.3	40.2	16.3	43.7	9.9	19.3	71.7
16	1604.296	100.4	40.1	14.5	44.2	8.5	18.5	71.8
17	1704.587	100.3	40.6	15.4	44.2	9.4	17.5	73.5
18	1804.939	100.4	40.1	14.6	45	7.7	17.9	74.3
19	1905.616	100.7	38.2	19.2	42	13	16.7	70
20	2006.298	100.7	42.7	14.4	41.7	9.6	18.5	71.1
21	2106.581	100.3	40.1	17	42	8.8	21.1	69.9
22	2207.267	100.7	32.8	13.7	53.3	9.6	16.3	73.9
23	2307.888	100.6	40.3	15	45.8	9.3	17.7	74.6

2. Select **Insert->Line** and then select a Line style. The data will be graphed overtime but the X-Axis will be incorrect.

3. 

4.

3. Right click on the graph, select **Select Data…**, and click on the **Edit** button in the **Horizontal (Category) Axis Labels** field.

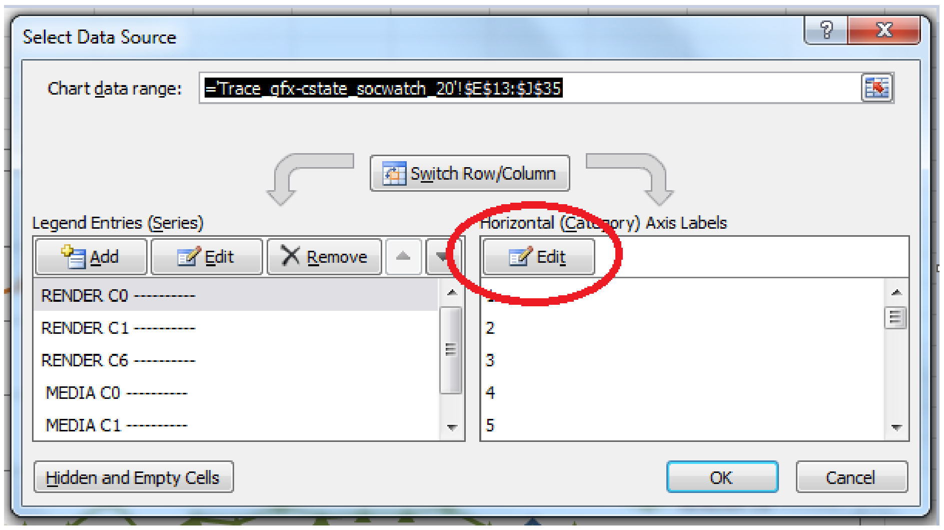
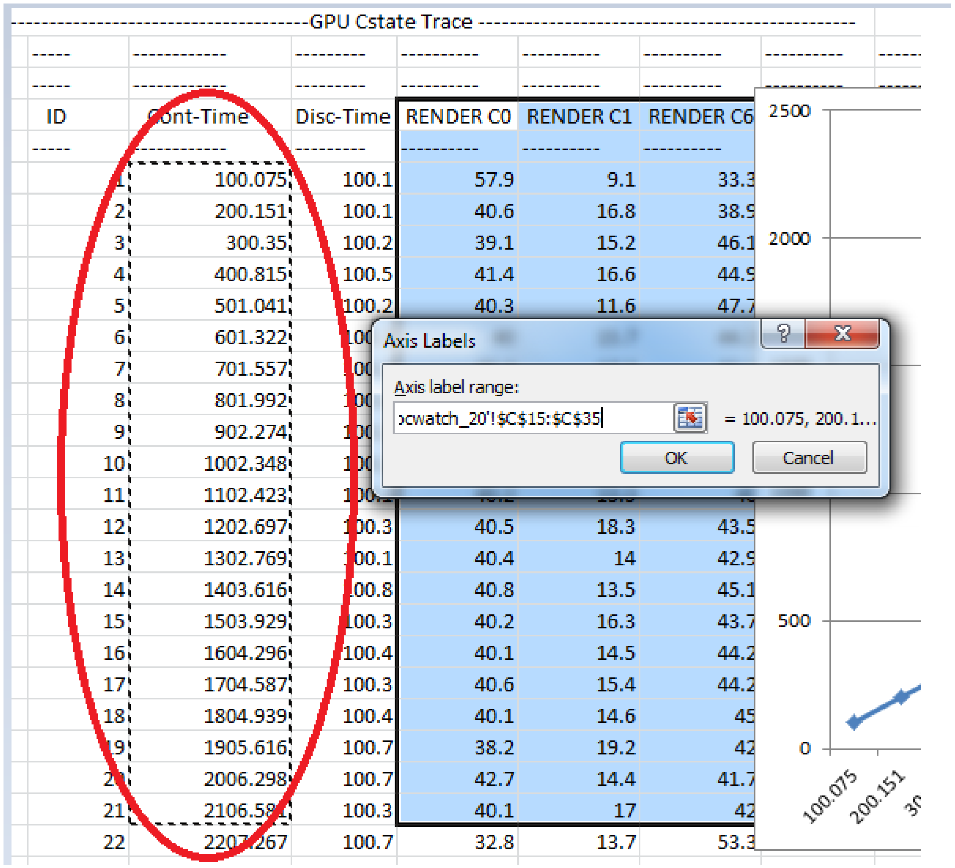

Select Data Source

Chart data range: ='Trace_gfx-cstate_socwatch_20'!$E$13:$J$35

Legend Entries (Series)	Horizontal (Category) Axis Labels
RENDER C0 ----------	
RENDER C1 ----------	2
RENDER C6 ----------	3
MEDIA C0 ----------	4
MEDIA C1 ----------	5

4. The **Axis Labels** dialog box will pop up.
5. Select the **Cont Time** data points that correlate with the data being graphed.

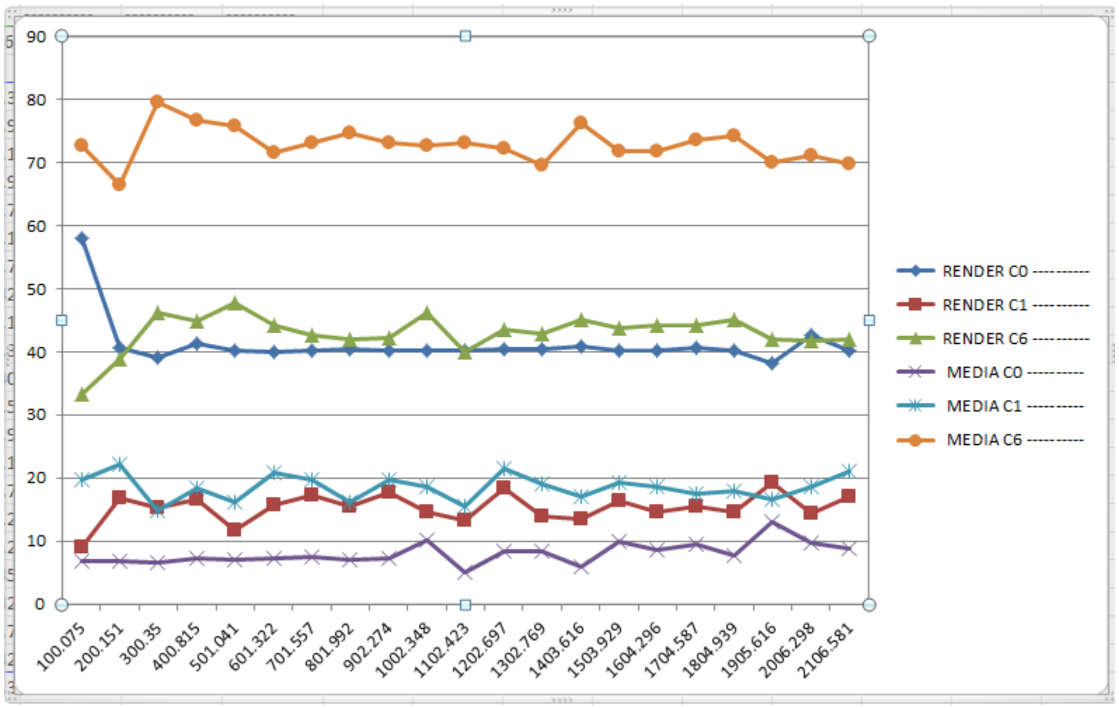

-------------------------------------GPU Cstate Trace ---

ID	Cont-Time	Disc-Time	RENDER C0	RENDER C1	RENDER C6
-----	------------	---------	----------	----------	----------
1	100.075	100.1	57.9	9.1	33.3
2	200.151	100.1	40.6	16.8	38.9
3	300.35	100.2	39.1	15.2	46.1
4	400.815	100.5	41.4	16.6	44.9
5	501.041	100.2	40.3	11.6	47.7
6	601.322				
7	701.557				
8	801.992				
9	902.274				
10	1002.348				
11	1102.423				
12	1202.697	100.3	40.5	18.3	43.5
13	1302.769	100.1	40.4	14	42.9
14	1403.616	100.8	40.8	13.5	45.1
15	1503.929	100.3	40.2	16.3	43.7
16	1604.296	100.4	40.1	14.5	44.2
17	1704.587	100.3	40.6	15.4	44.2
18	1804.939	100.4	40.1	14.6	45
19	1905.616	100.7	38.2	19.2	42
20	2006.298	100.7	42.7	14.4	41.7
21	2106.58	100.3	40.1	17	42
22	2207.267	100.7	32.8	13.7	53.3

Axis Labels

Axis label range: ocwatch_20'!$C$15:$C$35 = 100.075, 200.1...

5. Finally, select **OK** on the **Axis Labels** dialog box and **OK** on the **Select Data Source** dialog box to finish graphing the data.



Advanced Analysis of Raw Data

You can use the `socwatch_advanced_analysis_v1.4.py` python script to generate advanced analysis of the raw trace text data. The script can output the result as a TXT or CSV file depending on the script options utilized. If neither format is specified, it defaults to generating a textual summary to the console.

Use the following command to generate a textual summary of the results (assuming the `SoCWatchOutput.txt` file was generated by SoC Watch earlier using the `-r raw` switch):

```
> ./socwatch_advanced_analysis_v1.4.py -f ./SoCWatchOutput.txt --txt
```

The result will be written to the file `advanced_analysis.txt` in the current directory. Use the `-o` switch to specify an alternate output file name.

`socwatch_advanced_analysis_v1.4.py` supports the following options:

`-h, --help` Print an informative help message.
`-c, --core` Add per core data to the summary.
`--csv` Print the results in CSV format to an `advanced_analysis.csv` file in the current directory. To specify an alternate output file name, use the `-o` option (see below). This switch is incompatible with the `--txt` switch.
`--txt` Print all information in TXT format to an `advanced_analysis.txt` file in the current directory. To specify an alternate output file name, use the `-o` option (see below). This switch is incompatible with the `--csv` switch.
`-o, --output-file <output_file_name>` Use the supplied name to create the output files. `<output_file_name>` may optionally include a path; this path will be created if it does not already exist.

Advanced Analysis Data Description

The `socwatch_advanced_analysis_v1.4.py` script is capable of generating the following statistics:

- overall number of wakeups
- overall wakeups/second/core
- overall number of wakeups by timer, by interrupt, by IPI, by delayed workqueue execution, by the scheduler, or by unknown source
- overall percentage of wakeups by timer, by interrupt, IPI, by delayed workqueue execution, by the scheduler, or by unknown source
- overall number of C-state demotions
- overall percentage of C-state demotions

  NOTE: C-state demotions are defined as cases where the actual C-state was less than the requested C-state when the requested C-state was C6 or below.

- overall number of C-state aborts
  An abort occurs when the mwait instruction is executed but the core does not enter a C-state.

- overall C-state residency percentages for the system
- C-state residency percentages per core, per module, and per package
- overall P-state residency percentages
- P-state residency percentages per core

Two P-state residency tables are provided:
  - Hardware P-state residency table that provides the true P-state residency as seen by the hardware. This table takes into account the P-state when a core is in a C-state.
  - OS Requested P-state residency table that provides the P-state residency seen by the OS.

- C-states count per Wakeup Reasons
- percentage of C-state instances and residencies that wasted power or wasted an opportunity to save power in each sleep state and over all sleep states

- overall top 10 causes of wakeups
- overall top 10 interrupts causing wakeups
- overall top 10 processes causing wakeups via timers
- overall histogram of idle time broken down by idle duration
- overall wakeup metrics
- overall histogram of short sleep times followed by wakeups broken down by sleep time duration
- overall histogram of wakeups followed by short active execution (C0) times broken down by C0 time duration

- kernel wakelocks count per process histogram broken down by lock time duration
- kernel and user wakelocks lock/unlock summary

Note: Actual statistics generated will vary depending on the specific switches passed to SoC Watch when generating collection data.

Incorrect C-States Used

A transition into a C-state *wastes* power if the processor does not sleep long enough to amortize the energy cost of entering and exiting the C-state. The `socwatch_advanced_analysis_v1.4.py` script compares the actual residency of each C-state transition against the C-state target residencies published in the Linux source file `intel_idle.c`. If the C-state transition's actual residency is less than the C-state's target residency, the transition wasted power. For example, suppose an Intel® Atom™ processor core has to sleep 80µs or 128,000 cycles on a 1.6GHz processor in C2 before it starts to save power (that is, the overhead of entering and exiting C2 is the same power cost of running in C0 for 128,000 cycles). Therefore, if a core enters C2 and sleeps for less than 128,000 cycles, it wastes power. The entire sleep residency of each such instance is accumulated and divided by the total residency time to calculate the `Res Count (%)` metric for `Wasted Power Metrics` as shown in the table below.

```
----------------------------------------------------------------------------------------
INCORRECT C-STATES USED
----------------------------------------------------------------------------------------

C STATE        Wasted Power Metrics          Opportunity Lost Metrics      % Poor Dec

               Instances(%)   Res Count(%)   Instances(%)   Res Count(%)

*  C2          9.92           1.01           54.91          81.62          64.83
*  C4          47.07          10.64          50.1           87.69          97.17
*  C6          6.35           0.32           -              -              6.35

 Totals        12.24          0.68           31.84          9.21           44.08
(over all sleep states)
```

On the other hand, a transition into a sleep state wastes *an opportunity to save power* if its C-state residency is long enough such that it meets the target residency requirement to enter the *next* deeper C-state. For example, suppose an Intel Atom processor core has to sleep 400µs or 640,000 cycles on a 1.6GHz processor in C4 before it starts to save power (that is, the overhead of entering and exiting C4 is the same power cost of running in C0 for 640,000 cycles). Therefore, if a core enters C2 and sleeps for more than 640,000 cycles, it wastes an opportunity to save even more power (that is, sleeping in C4 saves more power than sleeping in C2 assuming the core sleeps long enough). The C-state residency of each such instance is accumulated and divided by the total residency time to calculate the `Res Count (%)` metric for `Opportunity Lost Metric` as shown in the table above.

C-States vs. Wakeup Reasons

The C-States vs. Wakeup Reasons table data breaks down the different reasons C-states are exited during a collection. For example:

```
        *Total*     Timer     Interrupt          IPI        WQExec      Scheduler        Unknown
*  C1       13         1             8             0             0              0              4
*  C1i     150         7           120             0            13              1              9
*  C2       46        36            10             0             0              0              0
*  C4       24        24             0             0             0              0              0
*  C6      384       260            66             0             0              0             58
```

Histogram of Idle Time

The Histogram of Idle Time table buckets each C-state transition's sleep time to help you quickly determine if the majority of sleep time is spent in longer durations (good) or shorter durations (bad). The initial buckets are delineated by the target residency thresholds for each C-state for the utilized processor architecture. For an Intel Atom processor, the bucket for C2 ranges from the target residency threshold for C2 to the target residency threshold for C4. Note that the lower bound for the C1 bucket is set to 0. The example table below has been reduced to fit properly in this document. The rows representing percentages will not sum to 100% because some table columns have been deleted to allow the table to fit into the document.

```
--------------------------------------------------------------------
HISTOGRAM OF IDLE TIME
--------------------------------------------------------------------


Header (ms)           0.0-0.08   0.08-0.4   0.4-0.56    0.56-1     1-5      5-10
                          C1        C2          C4         C6

Idle Bucket Time (us) 1263969     2638283      3184      133769  183406    466825

% Of Total Idle Time       5.4       11.28      0.01        0.57    0.78       2.0
Idle Bucket Hit Count   20607       20045         7         169      64        56
% of Total Idle Count    50.2       48.83      0.02        0.41    0.16      0.14
Total Idle Time (ms)    23393
Total Idle %             19.5
Avg. Idle Interval(us) 569.89
```

The `Idle Bucket Time` metric sums the total C-state transitions whose sleep time was within the bucket limits. For example, the above table shows that the sum of the time spent by all C-state transitions whose sleep time was between 0.4 and 0.56 ms is 3184µs. Note that summing all of the Idle Bucket Time values (that is, the Total Idle Time value) can result in a value greater than the collection time since the sleep time is counted for each core (assuming the system has more than one core).

The `% Of Total Idle Time` metric provides the percentage of the total time contained by a particular bucket. In the above example, 0.01% of the total idle time was spent in C-state transitions whose sleep time was between 0.4-0.56ms.

The `Idle Bucket Hit Count` metric provides the total C-state transition instances summed in the Idle Bucket Time metric. For example, 7 C-state transition sleep times were summed to 3184µs in the 0.4-0.56ms bucket.

The `% of Total Idle Count` metric provides the percentage of total C-state transition instances (that is, wakeups) summed in the Idle Bucket Time metric. For example, 0.02 % of the total C-state transitions had sleep times between 0.4-0.56ms in the above example.

The `Total Idle Time` metric is the total time, in milliseconds, cumulatively spent by all the cores in any sleep state. Note that the Total Idle Time value can result in a value greater than the collection time since the sleep time is counted for each core (assuming the system has more than one core).

The `Total Idle %` metric represents the percentage of time spent by the processors in a sleep state. For example, the above example shows that the cores spent 19.5 % of their time sleeping.

The `Avg. Idle Interval` metric value represents the average time, in microseconds, each core slept during each C-state transition (`Total Idle Time/Total Idle Hit Count`).

WakeUp Metrics

The Wakeup Metrics table includes data about specific causes of wake-ups. This table reports several metrics attributable to a specific Wakeup Reason in each row. In order, the columns following the Wake-up Reason represent the percentage of wakeups, the total wakeup count, the average wake-ups/sec per logical processor, the percentage trace coverage, and the CPUs that handled the wakeup. Trace coverage percentage is the percentage of the total collection time that experienced the wakeups across all CPUs involved in the collection.

For example, suppose a collection lasts for 100 seconds. If a Wakeup Reason occurred during the first second and the last second of the collection, it would have a TraceCoverage of 99%. If a Wakeup Reason occurred only between seconds 50 and 60 of the collection, it would have a TraceCoverage of 10%.

```
--------------------------------------------------------------
WAKEUP METRICS
--------------------------------------------------------------


Total # of wakeups :43
        Wakeup Reason          WU %      WU count     WU Rate(WU/sec/core) TraceCoverage(%)    CPUs

[PID] 17633 plugin-containe    6.98        3           0.95                  0.35               0+2+3
[IRQ]    54 eth0               4.65        2           0.63                  0.0                0+4
[PID]  1691 gnome-terminal     2.33        1           0.32                  0.0                0
[PID]     0 swapper            2.33        1           0.32                  0.0                1
```

Short Sleeps, Frequent Wakeups

The Short Sleeps, Frequent Wakeups table helps you find wakeup causes that occur after short periods of sleep. These wakeups should be minimized to allow a processor to enter a deeper C-state for longer periods of time, which in turn will minimize power consumption by the system. This table is sorted in the descending order by the total number of instances of each wakeup reason. The example table below has been trimmed and formatted to fit properly in this document.

```
--------------------------------------------------------------
SHORT SLEEPS, FREQUENT WAKEUPS
--------------------------------------------------------------
Wakeup Reason                                     Counts per Sleep Time Range(msec)

                               0.0-0.08   0.08-0.4   0.4-0.56   0.56-1       CPUs
                                  C1         C2         C4         C6

[IPI    IPI] IPI                  20        401        372        672         0+1
[IRQ     85] pvrsrvkm             17        174         64        125           0
[PID      0] swapper               0        146        115        196           0
```

The first column describes a specific wakeup reason. For a given wakeup reason, the various columns show counts of the number of times a processor was in a C-state for the time duration represented by that range. For example, in the table above, IRQ 85 woke up a processor 174 times after it was in a C-state between 0.08 and 0.4 milliseconds.

Short C0 Time, Frequent Wakeups

The Short C0 Time, Frequent Wakeups table is complimentary to the Short Sleeps, Frequent Wakeups table. The Short C0 time, Frequent Wakeups table describes how often the processor stays in C0 for a range of time after a specific wakeup reason. The information in this table is sorted by the total number of instances of each wakeup reason.

```
--------------------------------------------------------------
SHORT C0 TIME, FREQUENT WAKEUPS
--------------------------------------------------------------
      Wakeup Reason                     Counts per C0 Time Range (msec)

                          0.0-0.08         0.08-0.4         0.4-0.56         0.56-1   1-5      CPUs


[PID 2276]bw              5574             21675            696              11695    496      0+1
[PID    0]swapper         8                79               11               3        2        1
[IRQ   11]i2c-dw-pci-1    1                0                0                0        0        0
```

Each column in the above table represents a range of active execution (C0 time), following a wakeup. For a given wakeup reason, the various columns count the number of times the system was active for the time duration represented by that range. This is particularly useful in identifying periodic wakeups that cause the system to break out of idle for very short time periods before going back to sleep. Such wakeups would have high counts in the smaller time range columns and should be avoided because they waste power by preventing the system from entering deep sleep states. For example, in the table above, there were 5574 instances of the process `bw` causing a wakeup that lasted less than 0.08 milliseconds.

Kernel Wakelock Count per Process

The Kernel Wakelock Count per Process histogram contains a list of all processes that held a wakelock during the collection. The counts for each process are broken down by the duration in milliseconds of the locks the process held. The data below illustrates typical wakelock data, and has been formatted to fit this document properly.

```
-------------------------------------------------------------
KERNEL WAKELOCK COUNT PER PROCESS
-------------------------------------------------------------

Locking Process                              Lock Counts per Time Range(msec)


PID      PNAME              0-1      1-5      5-30    30-50    50-100    100-500    500-1000    1000-5000

332      InputReader         3257     115        9     0        0          0          0             6
0        swapper             1585     341        2     2        1          0          0             0
53       irq/318-mxt224      1081      28        1     0        0          0          0             0
150      irq/258-wl12xx       111     737        1     0        0          0          0             0
138      mediaserver          640      17       19     8        0          0          0            15
8        kworker/0:1          183       2        3     0        0          0          0             0
21       kworker/0:2          171       0        0     0        0          0          0             0
```

The `PID` and `PNAME` column give the ID and name respectively of the process that held the wakelock(s). The rest of the columns represent the duration(s) of the wakelocks held by the process. For example, the row with `PNAME irq/318-mxt224` and `PID 53` held 1081 wakelocks with a duration between 0-1 milliseconds, 28 wakelocks with a duration between 1-5 milliseconds, 1 wakelock with a duration between 5-30 milliseconds.

Kernel Wakelocks Summary

The Kernel Wakelocks Summary table includes a summary of the wakelocks locked and unlocked during the collection. For every process that held a kernel wakelock, the table shows the number of wakelocks held by that process, the total time for which wakelocks were held and the wakelock names. The data below illustrates typical wakelock data, and has been formatted to fit this document properly.

```
-------------------------------------------------------------
KERNEL WAKELOCKS SUMMARY
-------------------------------------------------------------

Locks acquired before collection start

ctp_charger_wakelock
0000:00:02.3
main

PID  No. of  Total Locked    TID   Process Name     Lock Name     Time(msec)    Op.    Unlock Process
     WLocks  Duration(msec)                                                             (PID:TID)

151  4       2               151   irq/258-wl12xx   rx_wake       1.0           L+U    Timeout
                             151   irq/258-wl12xx   rx_wake       0.83          L+U    Overwritten by:  Self
                             151   irq/258-wl12xx   rx_wake       1.0           L+U    Timeout
```

```
                               151   irq/258-wl12xx   rx_wake       1.0           L+U     Timeout

140   22      9364.53          1186  Binder_2         ApmCommand    2.98          L+U     ApmCommand (140:180)
                               324   AudioOut_2       AudioOutLock  3132.29       L+U     Self
                               324   AudioOut_2       intel_scu_ipc 0.14          L+U     Self
--- <snip> ---
```

The PID column gives the identifier of the process that held the wakelock(s), while the No. of WLocks column details the number of wakelock instances held by that process and the Total Locked Duration column represents the sum of the non-overlapping time in which the wakelock was held by that process. For example, in the table above, PID 151 held 4 wakelocks for a total of 2 milliseconds during the collection.

The TID column describes the thread identifiers of the various threads within that process, described by the Process Name column that held the wakelocks named in the Lock Name column. For example, for PID 151, 4 locks were held, one by TID 151. The Unlock Process (PID:TID) column indicates the process name, PID, and TID that unlocked the wakelock. Note that Timeout indicates the wakelock was unlocked automatically due to an expiring timer. Self indicates that the wakelock was unlocked by the same process that locked it. Overwritten by: Self indicates that the wakelock was relocked by the original locking process.

The Time column shows the individual lock time for each lock held. Note that a process might just lock or just unlock a lock during the collection, meaning the measured lock held time must be a smaller value than the actual lock held time. In this situation, if the lock was just locked during the collection, the Time value is calculated as (Collection End TSC – Lock TSC). Similarly if a lock was just unlocked during the collection, the Time value is calculated as (Unlock TSC – Collection Start TSC).

Corresponding to this, the Op. performed during collection column (abbreviated as the Op. column in this document) shows the operation that was performed on the Lock during the collection. L signifies the wakelock was Locked (acquired) during the collection, but was not released before the collection ended. U means the wakelock had already been acquired before the collection began and was then Unlocked (released) while the collection was ongoing. And finally, L+U signifies that the given wakelock was both acquired and released during the collection.

User Wakelocks Summary

The User Wakelocks Summary table includes a summary of the wakelocks locked and unlocked during the collection. For every application/package that held a wakelock, the table shows the User ID (UID) assigned by Android* to the application/package, the process name, the number of wakelock instances held by that application, the total time for which wakelocks were held, the associated PID, wakelock type, the wakelock tag, the time a specific wakelock was held, the operations performed on the lock (lock, unlock, or both), and the PID that unlocked the lock. The data below illustrates typical user wakelock data, and has been formatted to fit this document properly.

```
UID   Package        No. of  Total Locked    PID   Wakelock     Lock Tag                Time(msec)   Op.   Unlock
      Name           WLocks  Duration(msec)         Type                                                   PID

1000  system server  9       7520.65         299   PARTIAL      ActivityManager-Launch  141.52       L+U   299
                                             299   PARTIAL      ActivityManager-Launch  517.02       L+U   299
                                             299   SCREEN_BRIGHT KEEP_SCREEN_ON_FLAG    2275.14      L+U   299
                                             299   PARTIAL      AlarmManager            52.34        L+U   299
                                             299   PARTIAL      AlarmManager            3.38         L+U   299
                                             299   SCREEN_BRIGHT KEEP_SCREEN_ON_FLAG    4487.55      L+U   299
                                             299   PARTIAL      AlarmManager            32.92        L+U   299
                                             299   PARTIAL      *vibrator*              63.47        L+U   299
```

```
                                                      299   PARTIAL     ActivityManager-Launch 106.15       L+U   299

10075 com.antutu.ABenchMark 1 33837.55                5330  SCREEN_BRIGHT ABenchMark           33837.55     L+U   5330

1013  media server    3       9441.23                 140   PARTIAL     AudioOut_2             3161.26      L+U   140
                                                      140   PARTIAL     AudioOut_2             3118.04      L+U   140
                                                      140   PARTIAL     AudioOut_2             3161.93      L+U   140
```

SoC Watch Tips

If the collection needs to be performed untethered (that is, not connected to a host via a USB cable), use the `nohup` command with a start delay when launching SoC Watch and then disconnect the USB cable. The `nohup` command causes the terminal to ignore the hangup signal which would normally log the user out of the terminal when the USB cable is pulled. For example, use the following command to launch a 10 second collection after a start delay of 5 seconds. During the 5 second start delay, disconnect the USB cable from the device.

```
> nohup ./socwatch -s 5 -f cpu-cstate -t 10 &
```

Avoid using a CTRL-C via "adb" to terminate SoC Watch collections when running from Cygwin on a Windows host. Instead, use a timed collection (with the `-t` switch). Alternatively, use a Linux host.

If you see C-state wakeups caused by timers with either the thread ID or the process ID set to -1, then double-check your kernel parameters: this sometimes occurs if the kernel was not compiled with the `CONFIG_TIMER_STATS=y` option. In addition, under certain (rare) conditions, it is possible to have a wakeup caused by a timer for which SoC Watch captures a thread ID but not a process ID. In this case, SoC Watch assigns the thread ID to the process ID, and the process name to `Unknown-Process-Name`.

To submit a SoC Watch for Android bug, request group access through IEM2 (http://iem2.intel.com/Default.aspx). Select *Request Group Access*. Enter *ssg-dpd-bugzilla* and click on *Check*. Fill in the appropriate fields and submit your request. When your request is approved, you will receive a confirmation email. Finally, submit bugs at https://ssg-bugzilla2.fx.intel.com:8443/bugzilla/. If your issue describes a failed collection or a failure to post process raw SoC Watch results, please attach the SW1 file associated with the issue to the BZ.

General question about SoC Watch should be submitted to the Planet Blue SoC Watch Users Group at http://planetblue.ith.intel.com/gethelp/socwatch_users_group/default.aspx.

IPI Wakeups

Occasionally, interrupt, timer, workqueue execution, or sched_wakeup tracepoints do not occur before a core is placed again in a C-state. In other words, the C-state entry tracepoint occurs twice without an interrupt, timer, workqueue execution, or sched_wakeup tracepoint occurring in between. The associated wakeup may be caused by an IPI or Inter Processor Interrupt. IPIs are sent from one processor to another for different reasons[1]. SoC Watch monitors some per-CPU kernel IRQ statistics[2] at the same time as the TSC, MPERF, and C-state residency MSRs are read. If the IPI counts increment during a C-state sample and an interrupt, timer, workqueue execution, or sched_wakeup tracepoint did not occur, the sample's break type is set to IPI. Experiments were done with patched kernels[3] to confirm the validity of this strategy.

[1] See http://en.wikipedia.org/wiki/Inter-processor_interrupt.
[2] See the irq_stat struct in the struct_irq_cpustat_t struct in the arch/x86/include/asm/hardirq.h file.
[3] See the smp_reschedule_interrupt and smp_call_function_interrupt functions in the arch/x86/kernel/smp.c file and the local_apic_timer_interrupt function in the arch/x86/kernel/apic/apic.c file.

Appendix A: Acronyms and Definitions

ACG: Autonomous Clock Gating

ACPI: Advanced Configuration and Power Interface

AON: Always On

DDR: Double Data Rate SDRAM

DVFS: Dynamic Voltage Frequency Scaling

FW: Firmware

LSS: Logical SubSystem

MCG: Mobile Computing Group

MSIC: Mixed Signal Integrated Circuit

MSR: Model-Specific Register

PMIC: Power Management Intergrated Circuit

PSI: Platform and System Integration

PSS: Physical SubSystem

SCU: System Control Unit

SDRAM: Synchronous Dynamic Random-Access Memory

TSC: Time Stamp Counter