Intel® Energy Checker

SDK Device Driver Kit User Guide

Revision 2.0

December 15, 2010

Revision History

Revision	Revision History	Date
1.0	Initial Release	2009/10/15
2.0	2005.12.15 Release	2010/12/15

Contents

Figures

Tables

1 Introduction

This *Intel® Energy Checker Device Driver Kit User Guide* is part of the Intel® Energy Checker (Intel® EC) Software Development Kit (SDK). The Intel EC SDK enables developers of Independent Software Vendors (ISVs) to easily import and export counters in their source code. Although the initial intent of the Intel EC SDK is to facilitate software energy efficiency analysis and optimizations, it can be used to expose any counter meaningful to each ISV and its customers.

The Intel® Energy Checker is also referred to as the Intel® EC. The Intel EC SDK is also referred to as the SDK. The Intel EC Device Driver Kit is also referred to as the Device Driver Kit and Kit.

Detailed descriptions of the SDK and use models can be found in the *Intel® Energy Checker Software Developer Kit User Guide*.

NOTE

This user guide assumes the reader is familiar with the 64-bit counters and Productivity Links (PLs) described in the Intel® Energy Checker SDK User Guide. Please refer to that document for a description of counters and PLs.

This *Intel® Energy Checker SDK Device Driver Kit User Guide* describes two key server applications provided with the Intel® EC SDK and how they can be extended to support additional devices:

- The Energy Server (ESRV) measures energy consumption and power usage of instrumented servers or using external power meters.
- The Temperature Server (TSRV) measures temperature and relative humidity via supported sensors. Energy efficiency results with similar systems can vary depending on critical environmental parameters. The TSRV helps establish baseline conditions for comparative purposes and long-term trending analysis.

ESRV and TSRV have similar structures and command-lines to make it easier for developers and end-users to integrate energy and temperature sensing into their applications, thereby facilitating the development of energy-aware software. Both utilities allow developers to extend the list of supported devices through the use of device libraries. The chapters in this User Guide cover the following topics.

- This Overview of the Device Driver Kit
- Overview and application of ESRV and energy monitoring devices
- Overview and application of TSRV

The Intel® EC SDK includes ESRV and TSRV binaries for 32-bit and 64-bit X86 processors running under Windows*, Linux*, Solaris* 10, MacOS*, and MeeGo* operating systems. The SDK also includes sample source code that can be used to access ESRV and TSRV counter data. Refer to the *Intel® Energy Checker SDK User Guide* for more information on supported languages and operating systems.

This page intentionally blank.

2 ESRV

The ESRV uses external power meters or specially instrumented power supplies to monitor the power consumption of

- a given server,
- group of servers,
- any arbitrary equipment the power meter or instrumented power supply monitors.

Monitored parameters include cumulative energy consumption, as well as sampled readings for voltage, current, power, and power factor correction. ESRV also provides a data acquisition (DAQ) mode, which allows the monitoring of system components, such as memory or IO subsystems.

ESRV measures energy in joules (J) and in kilowatt-hours (kWh).

NOTE

*Though they are often interchanged terms, an important distinction must be made between **power** and **energy**. Power is a **rate** of energy consumption (typically measured in watts (W) or kilowatts (kW)), while energy is a **cumulative measure of power over time** (often billed in kilowatt-hours (kWh)).*

NOTE

At a more granular level, energy is often measured in joules (J). One joule is equivalent to one watt-second. There are 3,600 joules (watt-seconds) in one watt-hour and 3,600,000 joules (watt-seconds) in one kWh.

Application developers can integrate ESRV data into their applications to directly influence software system operation or to determine the most energy-efficient algorithm for completing a given task with a given hardware configuration. Network administrators or system integrators can use ESRV data to determine what hardware configuration (or combination of hardware and software) provides the most efficient performance for a given application. This is much more accurate than relying on extrapolation from standard benchmarks to estimate the performance and efficiency of a specific application on the selected hardware/software combination. Additional use cases for Intel EC SDK instrumentation are outlined in the *Intel Energy Checker SDK User Guide*.

As shown in Figure 1 below, the PL GUI Monitor application (provided with the Intel EC SDK) can provide a graphical representation of ESRV data in real time.

NOTE

Section 2.5.10 describes the meaning of each ESRV counter.

Figure 1: ESRV counters in PL GUI Monitor

2.1 ESRV Hardware Setup

ESRV requires a connection to some device to measure energy consumption. This connection can be through a serial port, USB port, locally instrumented power supply, or some other supported interface. You can use ESRV to monitor the energy consumption of local systems or remote system(s). Figure 2 below shows an example configuration where ESRV is running on a system separate from the monitored system.

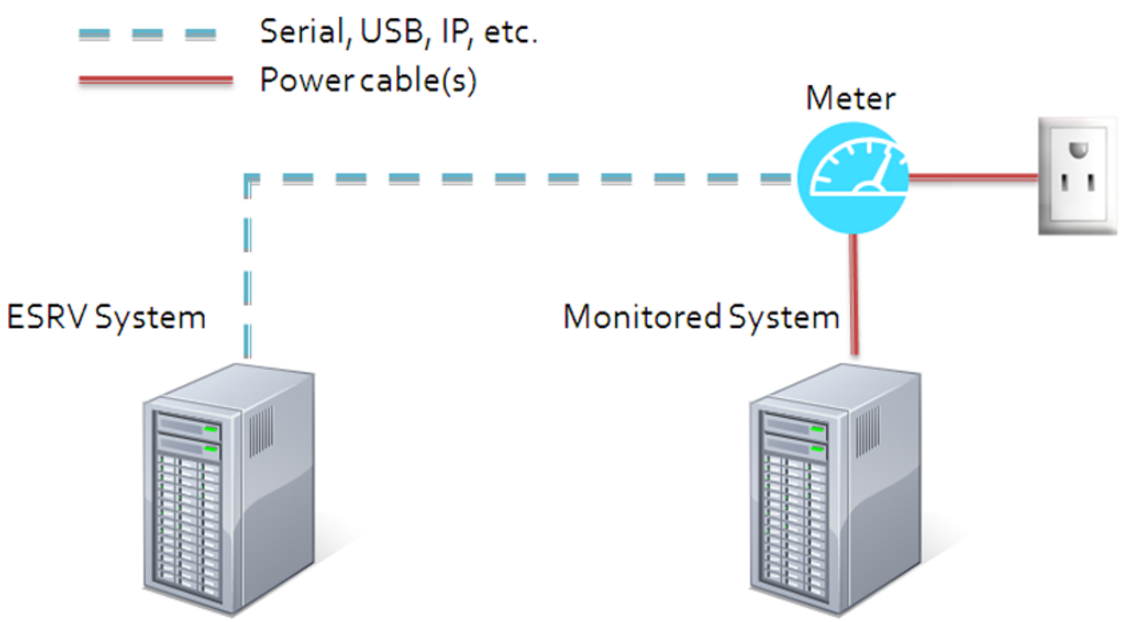

Figure 2: Example remote energy monitoring setup

NOTE

A single server can host multiple instances of ESRV, provided it has the required interfaces and measuring devices in the right quantity. However, when interrupted by the <CTRL>+<C> key combination, all instances of ESRV are stopped.

NOTE

If the metering hardware supports it, ESRV supports monitoring of systems with single-phase (1Φ), three-phase (3Φ), or direct current (DC) power input.

2.1.1 Measuring Power without Unplugging Servers

With the right equipment, ESRV can monitor servers without having to shut down or unplug the servers first. Figure 3 below shows an example configuration where a removable current clamp-on probe is used with an external power meter to measure the server's power usage.

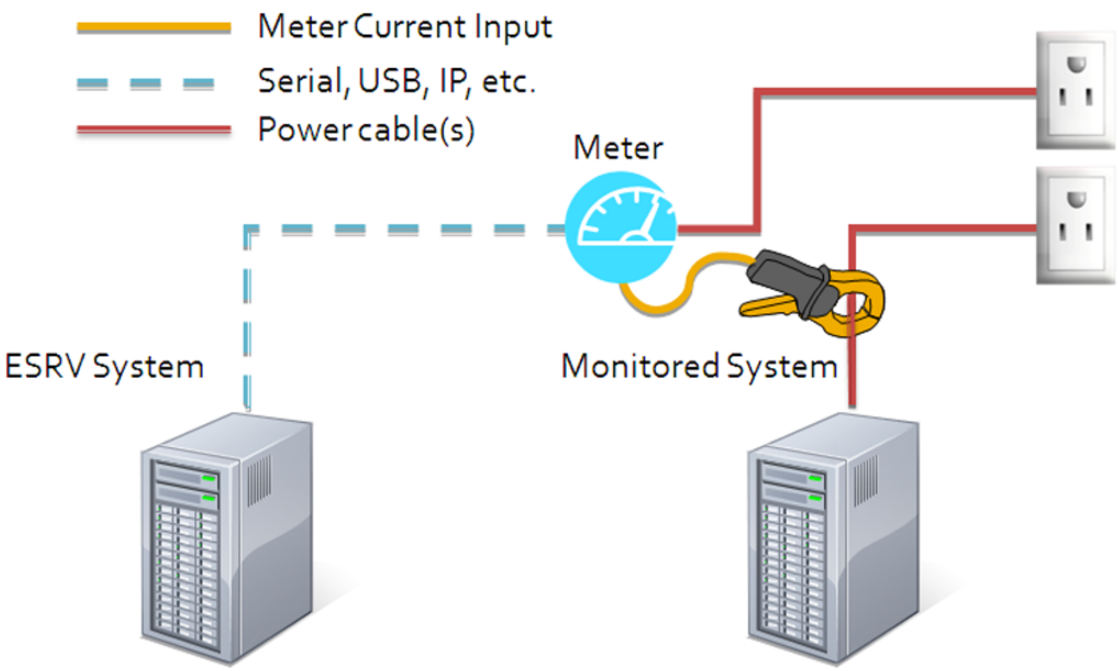

Figure 3: Example remote energy monitoring setup with clamp-on probe

The meter measures the current draw (in amps) via the clamp-on probe (sometimes called a "current clamp"). To monitor power, the meter also needs to measure the voltage. This can be accomplished by providing a connection from the meter to the electrical source, preferably on the same mains circuit as the monitored system.

Clamp-on probes are available from a wide variety of sources. Configurations like the one shown in Figure 3 can routinely achieve accuracy exceeding 98% (less than 2% error) and can achieve greater than 99% accuracy (less than 1% error) with some measurement equipment.

NOTE

If power supplies in the system being monitored are instrumented sufficiently, ESRV can monitor such a system without the need for an external meter and without the need to unplug or restart the system.

2.2 ESRV User Interface

ESRV is a command-line tool. You can perform the following operations:

- **start** an ESRV instance
- **stop** one or all ESRV instances on the server
- **reset** the energy counters of one or all ESRV instances on the server

Each command has a contextual help (`--help`).

2.2.1 Starting ESRV

The listing below provides the start command help message: `esrv --start --help`

```
Start energy server.

Usage:  esrv --start --device <dev_name> [--device_options <options>] [--
interface_options <options>] [channel] [--diagnostic] [--pause <t>]
Or      esrv --start --library <lib_name> [--device_options <options>] [--
interface_options <options>] [channel] [--diagnostic] [--pause <t>]
             [--offset_power <w>] | --offset_power_samples <s>] --offset_pause <t>

   dev_name is one of the following (case insensitive):
      "y210", "Yokogawa_wt210":  Yokogawa* WT210 external digital power meter
      "e801", "Extech_380801":  Extech* 380801 external digital power meter

   lib_name is the filename of a supplemental library for other meters
      The filename is usually a .DLL file in Windows or a .SO file otherwise
      Use quotes around lib_name if it contains spaces

   options (device) refers to the device options (delimited by quotes)
   Check the systems integrator or meter vendor's manual for details
      For example, "eor=lf" for the Yokogawa WT210 external digital power meter
      "y210", "Yokogawa_wt210":  [eor={cr|lf|crlf}]
      "e801", "Extech_380801":  At the current time, no options for this device

   options (serial) refers to the input interface options (delimited by quotes).
   Serial port option strings supported:

[com=c][baud=b][parity={E|O|N|I}][data={7|8}][stop={0|1|2}][xon={Y|N}][dtr={Y|N}][rts=
{Y|N}]
      baud={1200|2400|4800|9600|19200|38400|57600|115200|230400}
      For example, "com=0 baud=9600 parity=n data=8 stop=1"
      The default serial options are "com=1 baud=9600 parity=n data=8 stop=1 xon=n
dtr=y rts=n"

   channel is the interface to monitor on a multi-channel meter
      For example, to read the second channel on a 3-channel meter, use 2
      If channel is omitted, channel 1 is implied
      If channel is equal to 0, all channels is implied
      All counters are prefixed with "[CHANNELx] - ", where x is the channel number.

   --diagnostic activates diagnostic messages display during runtime.
   --pause refers to the server's sampling interval given in seconds.
      By default, pause is 1 second.

   --offset_power [w] do not account for power readings lesser than [watts].
   --offset_power_samples [s] computes the average power in Watts over [s] samples.
```

```
      by using the --offset_pause [t] option with the previous one, it is possible to delay
      the sampling by [t] samples.

   If the energy server is successfully invoked, then the following elements
   are available:
      A guid printed on the standard output. The guid is a globally unique
      identifier for this set of counters.
      (for example: 273735b1-8319-40fd-b48b-cb9718de415c).
      An "Energy (Joules)" counter.  A Joule equals one Watt-second.
      An "Energy (kWh)" counter.
      An "Energy (kWh).decimals" counter (indicating kWh is 100x the kWh reading)
      An "Energy Overflows (no unit)" counter.  Incremented each time
          the Energy in Joules counter overflows (18,446,744,073,709,550,615 J).
      A "Power (Watt)" counter (for the last power reading).
      A "Power (Watt).decimals" counter.
      An "Update Frequency (seconds)" counter. Set with the --pause option

   * Third-party trademarks are the property of their respective owners..
```

2.2.1.1. Example Command Lines

Here are four examples of how to start ESRV:

```
esrv --start --device y210 --device_options "items=all"
esrv --start --device e801 --interface_options "com=PL2303-000103D"
esrv --start --library yokogawa_wt230.dll --interface_options "com=2 dtr=n"
./esrv --start --library ./yokogawa_wt210.dll --interface_options "com=1 baud=9600 parity=n data=8 stop=0 xon=n dtr=y rts=n"
```

NOTE

Under MacOS X, driver writers can name their devices freely. This applies to serial ports, too. Therefore, the ESRV* `com` *option doesn't take a numerical argument, but takes the name of the cu device. Once the driver is installed on the system, search for its name in the* `/dev` *folder (*`ls -l /dev/cu.*`*). Use the name after the dot (.) for the com option. For example, if the USB-to-serial adapter is named* `/dev/cu.PL2303-000103D`*, then the option is* `com=PL2303-000103D`*.*

NOTE

ESRV may report that it is unable to open a given serial port. This generally happens when a few options of the default configuration are changed. For example, specifying a baud rate of 19200 and serial port number 5 (`--interface_options "com=5 baud=19200"`*). You might need to provide a fully qualified serial interface options string, which includes all serial parameters (port, baud, parity, data and stop bits, etc.). If ESRV fails to open the serial port, which is known to be functional and was used in the past with ESRV or another application, then try a fully qualified interface options string, like the following in the scope of our example:* `--interface_options "com=5 baud=19200 parity=n data=8 stop=1 xon=n dtr=y rts=n"`*.*

Figure 4 shows a sample output from ESRV after startup (MacOS).

```
Terminal — esrv — 91x38
DIAG: Setting-up Inter-Process Communication. [COMMON]
DIAG: Energy Server Starts. [SERVER]

*********************************************
* PL SDK Sample Energy Server: START        *
*********************************************

Sample Energy Monitor: esrv version 2009.03.03 - using PL version 2009.03.06(M)

DIAG: Loading And Configuring Device. [SERVER]
DIAG: ...Registering init extech 380801 extra data Function. [SERVER]
DIAG: ...Registering delete extech 380801 extra data Function. [SERVER]
DIAG: ...Registering open extech 380801 Function. [SERVER]
DIAG: ...Registering close extech 380801 Function. [SERVER]
DIAG: ...Registering parse extech 380801 option string Function. [SERVER]
DIAG: ...Registering read extech 380801 power Function. [SERVER]
DIAG: ...No read energy extech 380801 Function To Register. [SERVER]
DIAG: Setup Thread Synchronization Data. [SERVER]
DIAG: Parsing Interface Options. [SERVER]
DIAG: Initializing Device Extra Data - First Call. [SERVER]
DIAG: No Device Options String To Parse. [SERVER]
DIAG: Initializing Device Extra Data - Second Call. [SERVER]
DIAG: Opening Interface. [SERVER]
DIAG: ...Openning Serial Port. [SERVER]
DIAG: ...Configuring Serial Port. [SERVER]
DIAG: ...Purging Serial Port. [SERVER]
DIAG: Opening Device. [SERVER]
DIAG: Installing <CTRL>+<C> Signal Handler. [SERVER]
DIAG: Spawning Collector Thread. [SERVER]
DIAG: Starting Inter-Process Communication Controller. [SERVER]
DIAG: Waiting For User Command. [SERVER]
DIAG: ...Preparing PL Counters Data. [KL-SERVER]
DIAG: ...Opening PL. [KL-SERVER]
Using Guid: [BD2894A1-2A67-424C-BCA2-4E8F62BB4343]
DIAG: ...Using The Guid [BD2894A1-2A67-424C-BCA2-4E8F62BB4343]. [KL-SERVER]
DIAG: ...Writing Static Counters. [KL-SERVER]
DIAG: ...Channel [#1] Queried. [KL-SERVER]
Samples:    [106]
```

Figure 4: Typical Output of ESRV (MacOS X)

2.2.1.2. Advanced Startup Options

ESRV provides some advanced options that can be added to the command line when starting ESRV:

- --offset_power <w>
- --offset_power_samples <s>
- --offset_pause <t>

NOTE

These options only affect the reported power levels and reported energy consumption. Actual measurements of voltage and current are not adjusted by these offset power options.

The `--offset_power <w>` option *subtracts* a constant value (in watts) from all power readings. For example, to remove a baseline power draw of 150 W from the reported readings, include the following option on the command line:

```
--offset_power 150
```

The `--offset_power_samples <s>` option is an alternate way to specify the offset power level. This option directs ESRV to read a specified number of power samples, compute the average power consumption in those samples, and use this as the power offset level for subsequent readings.

This option can be used in place of the `--offset_power` option to dynamically determine the baseline power draw of a system before an application starts taxing the system. The average value determined with this option is printed to stdout. For example, to use the first 40 power readings as a baseline offset for subsequent power readings, include the following option on the command line:

```
--offset_power_samples 40
```

The `--offset_pause <t>` option specifies a time period (in seconds) for ESRV to wait before sampling power. When used in conjunction with the `--offset_power_samples` option, the `--offset_pause` option allows the system to reach a stable condition before baseline samples are taken. For example, to wait 30 seconds before using the first 40 power readings as a baseline offset for subsequent power readings, include the following options on the command line:

```
--offset_power_samples 40 --offset_pause 30
```

NOTE

The `--offset_power_samples` *option and the* `--offset_pause` *option delay ESRV energy and power reporting. If repeated tests are done on the same system, later invocations of ESRV can use the* `--offset_power` *option to start ESRV faster once the baseline power draw has been established for that system.*

NOTE

These options affect all channels on a meter with multiple channels.

Here are example command lines and the results printed to standard out.

```
esrv --start --device y210 --offset_power 150
```

```

********************************************
* IECSDK Sample Energy Server: START        *
********************************************

Sample Energy Monitor: esrv version 2009.03.03 - using PL version 2009.07.01(W)

Server Name:    [esrv]
Using Guid:     [bfda3be6-cb84-4918-a433-f307e2de918a]
Offset Power:   [Channel #0: 150.000000] Watt(s)
Samples:        [38]
```

```
esrv --start --device y210 --offset_power_samples 20 --offset_pause 10
```

```

*********************************************
* IECSDK Sample Energy Server: START        *
*********************************************

Sample Energy Monitor: esrv version 2009.03.03 - using PL version 2009.07.01(W)

Server Name:    [esrv]
Using Guid:     [7dee38c4-9fd7-47ec-b9f5-69ab1264a838]
Pausing For:    [10] Second(s) Before Starting Power Offset Data Collection
Power Samples:  [20/20]
Offset Power:   [Channel #0: 187.211000] Watt(s)
Samples:        [2]
```

2.2.2 Stopping ESRV

The listing below shows the stop command help message: `esrv --stop --help`

```

Stop energy server.

Usage:  esrv --stop [guid] [--diagnostic]

guid is the identification string displayed when esrv was started.
If guid is 0 or omitted, all active esrv session are stopped.
--diagnostic activates diagnostic messages display during runtime.
```

2.2.3 Resetting ESRV

The listing below shows the reset command help message: `esrv --reset --help`

```

Reset energy counters to zero.

Usage:  esrv --reset [guid] [--diagnostic]

guid is the identification string displayed when esrv was started.
If guid is 0 or omitted, all active esrv session are reset to zero.
Note: all the channels of the target ersv instance(s) are reset.
--diagnostic activates diagnostic messages display during runtime.
```

2.2.4 ESRV Ranges

ESRV provides additional help information with the `--ranges` command. If this command line option is used, ESRV prints out information on the range of energy that can be accumulated by ESRV before counters overflow. The listing below shows the output of the `esrv --ranges` command.

```

Display this message on supported data ranges.

Usage:  esrv --ranges

   Max Joules.....................................184,467,440,737,095,516.15
   Max kWh..............................................51,240,955,760.30
   Max kWh With Overflows..........945,228,797,002,527,336,667,005,373,644.80

                 Time to Overflow                          Rate
   ___________________________________________________________________________
   ..............5,849,424,173.55 years @......................1 W
   ................584,942,417.36 years @.....................10 W
   .................58,494,241.74 years @....................100 W
   ..................5,849,424.17 years @..................1,000 W......1 kW
   ....................584,942.42 years @.................10,000 W.....10 kW
   .....................58,494.24 years @................100,000 W....100 kW
   ......................5,849.42 years @..............1,000,000 W......1 MW
   ........................584.94 years @.............10,000,000 W.....10 MW
   .........................58.49 years @............100,000,000 W....100 MW
   ..........................5.85 years @..........1,000,000,000 W......1 GW
```

ESRV measures energy consumption in hundredths of a joule, but as can be seen from the information above, most environments are unlikely to overflow their energy counters with ESRV.

2.3 ESRV Data Acquisition Mode

By default, ESRV measures the energy consumed by one or more systems at the platform level – or power at the wall – using power meters or specially instrumented power supplies. In most cases this capability is sufficient to conduct an energy efficiency study, to optimize software for energy efficiency, or to add energy-saving heuristics to an application. However, you can also break down the platform's energy consumption per platform component using the Data AcQuisition, or DAQ, mode.

To use ESRV in DAQ mode, a DAQ module is required. ESRV supports the Yokogawa MW100 DAQ (see section 2.5.2), with a standard, built-in interface, so DAQ vendors can develop an ESRV support module (see section 2.5.13) for their equipment.

Always refer to your DAQ and board documentation for the setup, and apply features safely as recommended by the manufacturers.

Figure 5 shows a Yokogawa MW100 DAQ unit used to monitor the power draw and the energy consumed by four memory DIMMs in a server board. Special memory DIMM risers instrument the board. The wires connected to the acquisition module are soldered around a sense resistor on the server side of the riser.

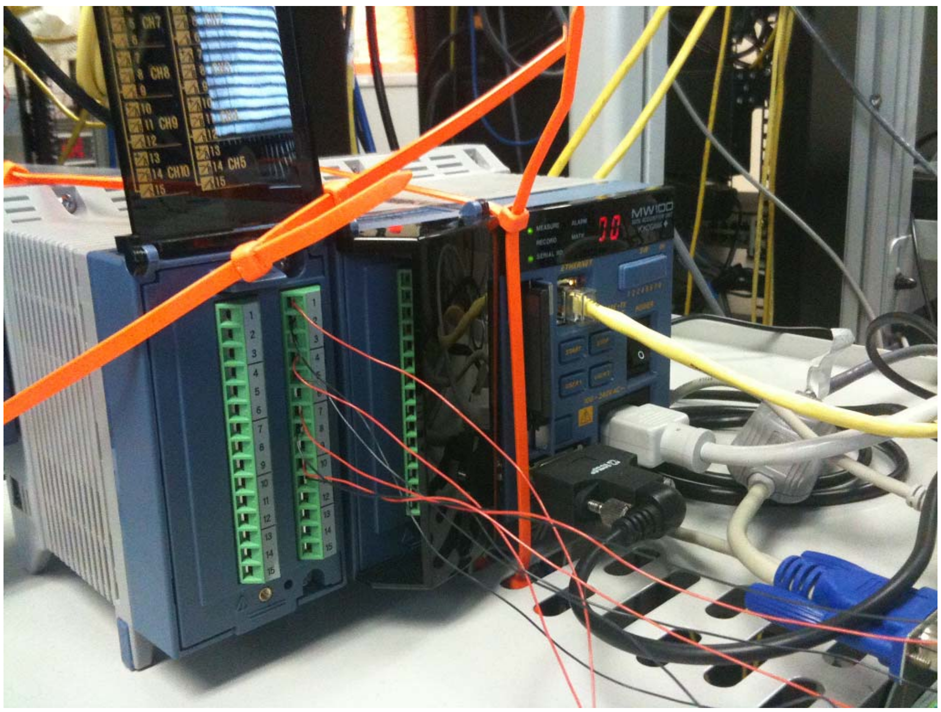

Figure 5: Yokogawa MW100 DAQ connected to an instrumented server platform's memory DIMMs

Using the setup above as an example, **pl_gui_monitor** (a companion application of the SDK) monitors the PL counters exposed by an ESRV instance running in DAQ mode and communicating with the MW100 (Figure 6). In this example, both applications (pl_gui_monitor and ESRV) run on a remote system under the Windows* operating system (OS). The *monitored* system (the instrumented server of the setup), runs the Linux operating system and is performing memory bandwidth benchmarks, selectively stressing the memory DIMMs.

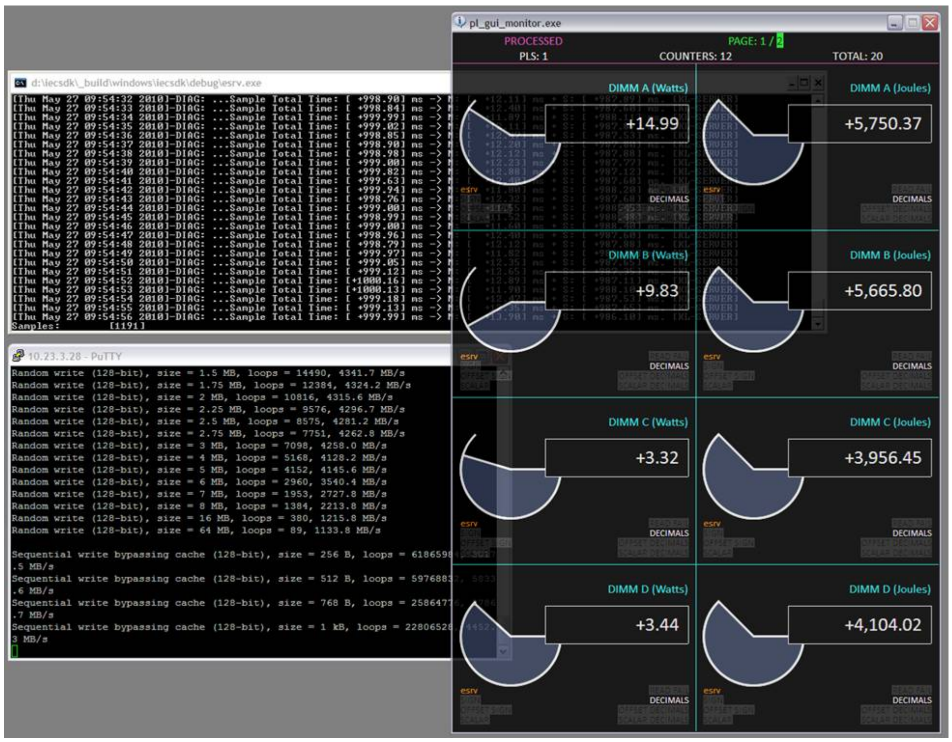

Figure 6: ESRV DAQ counters monitoring in pl_gui_monitor of a server's four (4) DIMMs' power draw and energy consumption

Figure 7 shows the memory power draw in the example, plus the energy consumed, monitored, and recorded in Microsoft* Perfmon.* The data conversion between the ESRV PL counters and the Windows native counters is performed using the pl2w utility of the Intel EC SDK (see the *Intel EC SDK User Guide* for details).

Figure 7: Memory DIMMs' power of a server running Linux monitored on a Windows system in pl_gui_monitor and plotted over time in Microsoft Perfmon (conversion done with pl2w)

NOTE

Similar to ESRV's default (power meter) mode, in DAQ mode, ESRV performs real-time reading of the DAQ channels. This is not the traditional two pass use of a DAQ (acquisition followed by post-mortem data processing).

NOTE

ESRV samples the DAQ channel counters at a low frequency – in comparison to sampling frequencies DAQs are usually capable of. However, if your DAQ allows continuous channel sampling and offers on-board math features, it is possible to benefit from DAQ high-frequency channel sampling and low-frequency ESRV sampling.

2.3.1 Starting ESRV in DAQ mode

The listing below shows the start command help message for the DAQ mode of ESRV:

```
esrv --daq --help
```

```
Start energy server in DAQ (Data AcQuisition) mode.

Usage:  esrv --start --daq --device <dev_name> [--device_options <options>]
         [--interface_options <options>] [--diagnostic] [--pause <t>]
         --channels <options>
         --counters <options>> | --counters_file <file> | --identity
         [--default_suffixes <options>]
         [[--time_integral] | [--integral]]
         [--kernel_priority_boost]
         [--math_threads <n>]
         [--math_threads_priority_boost]
         [--offset_counter <o> | [--offset_pause <t>] --offset_counter_samples <s>]
         [--instance_name <string>]


   dev_name is one of the following (case insensitive):
      "y100", "Yokogawa_mw100":  Yokogawa* MW100 data aquisition unit

   lib_name is the filename of a supplemental library for other data acquisition units
      The filename is usually a .DLL file in Windows or a .SO file otherwise
      Use quotes around lib_name if it contains spaces

   options (device) refers to the device options (delimited by quotes)
   Check the systems integrator or meter vendor's manual for details
      For example, "eor=lf" for the Yokogawa MW100 external data acquisition unit
      "y100", "Yokogawa_mw100":  [eor={lf|crlf}]

   options (serial) refers to the input interface options (delimited by quotes).
   Serial port option strings supported:

[com=c][baud=b][parity={E|O|N|I}][data={7|8}][stop={0|1|2}][xon={Y|N}][dtr={Y|N}][rts=
{Y|N}]
      baud={1200|2400|4800|9600|19200|38400|57600|115200|230400}
      For example, "com=0 baud=9600 parity=n data=8 stop=1"
      The default serial options are "com=1 baud=9600 parity=n data=8 stop=1 xon=n
dtr=y rts=n"

   --diagnostic activates diagnostic messages display during runtime.
   --pause refers to the server's sampling interval given in seconds.
      By default, pause is 1 second.

   options (channels) refers to the active channels of the data aquisition unit.
      channels' numbers are separated by spaces. Channel ranges are specified using
      a hyphen character between the lower and the upper bound (included)
   Example:
      --channels "1 2 5-10 18-15 50" to specify the following 13 channels:
         1, 2, 5, 6, 7, 8, 9, 10, 15, 16, 17, 18 and 50.

   Following options (3) defines the counters and how they are built using the
channels.

      --identity converts each channel into a counter. Each counter is named
      "DAQ Channel " with the channel number appended ("DAQ Channel 6", etc.).
      Integration can be applied to all channels by using the --integral option.

      file (counters) refers to the file used to define the counters and the equations
```

```
         used to compute counter value out of channel readings.
         The file is a plain ASCII file composed of a single line respecting the
syntax
         described in the last counter definition option (bellow).

      options (counters) refers to the definitions of the math equations used to
         compute the counters' value using channel readings. The definitions are
         expressed as a counter name followed by the equal sign (=) followed by
         a postfix** expression***. Each definition is separated by a comma (,).
         Note: TOS = Top Of the Stack, L1 = first level, L2 = second level, etc.
         Note: The evaluation stack has a maximum depth of 128 elements. Each
            element is a floating-point value (matches the long double C type).
         Note: Each expression must evaluate into a single value. If there are more
            or less values at the end of the evaluation then an error if raised.
         The postfix expression can be composed of:
            Channel readings (Cx where x is the channel number. C12 for channel 12,
etc.)
            Literals (1, -3.14, 1E-5, etc.)
            Operations:
               +: add the two first stack levels (TOS = TOS + L1)
               -: subtract the two first stack levels (TOS = TOS - L1)
               *: multiply the two first stack levels (TOS = TOS * L1)
               /: divide  the two first stack levels(TOS = TOS / L1)
                  Note: an error is raised if L1 = 0
            Functions:
               sign: push 1 (negative) or 0 (positive of null) to the stack
                  according to the sign of the TOS value (TOS=sign(TOS))
               ip: push the TOS value's integer part on the TOS (TOS=ip(TOS))
               fp: push the TOS value's fractional part on the TOS (TOS=fp(TOS))
               abs: push the TOS value's absolute value on the TOS (TOS=abs(TOS))
            Stack manipulation functions:
               dup: duplicate the TOS value (TOS -> TOS TOS)
               swap: swap TOS and L1 (TOS L1 -> L1 TOS)
               drop: remove the TOS value (TOS L1 -> L1)
            Counter qualifiers:
               integral: the computed value is summed into the counter
               offset: the counter can be offset (--offset* options)
               Note: a qualifier is not compiled and can appear anywhere in the
equation.
   Example:
      "RAID Energy (Joules) = C15 C16 C17 C18 + + + integral offset" defines a counter
      named "RAID Energy (Joules)" which is computed as the sum of channel 15, 16, 17
      and 18. This counter is an integral counter and can be offset.

    options (suffixes) refers to the default counter suffixes to be applied to all
counters.
       The following suffixes are recognized:
          sign
          decimals
          offset
          offset.decimals
          offset.sign
          scalar
          scalar.decimals
   Example:
      --default_suffixes "decimals = 2" will create a .decimals counter for
         each counter and sets it to the value 2.

   Used in conjunction with --identity, --integral sets all counters as integrals.
      The integration mode can be specified with --time_integral.

   --time_integral weights the counters value with the sample duration in seconds.
      By default, the integration is not weighted by sample interval time.
```

```

   --kernel_priority_boost increases math threads' priority.
      This option may require higher privileges than simple user.

   --math_threads <n> requests the use of <n> threads to compute the counters.
      This request is only satisfied if the number of counters justifies the
      cost of creating and managing the math threads.

   --math_threads_priority_boost increases math threads' priority.
      This option may require higher privileges than simple user.

   --offset_counter <o> do not account for counter values lesser than <o>.
   --offset_counter_samples <s> computes the average counter values over <s> samples.
      by using the --offset_pause <t> option with the previous one, it is possible to
delay
      the sampling by <t> samples.
      offset does not apply to integral counters.

   string following --instance_name is appended to esrv as the PL application name.
      This option solves the problem introduced by the --daq mode and pl2w Windows*
      registry pollution that may appear when the same application has different
      counters.  Since in --daq mode, the counters are freely defined by the user,
      this may very well happen.  Take special care not converting esrv PLs with pl2w
      when they have different counters!

   Example:
      esrv --start --daq --channels "1 2 5-10 18-15 50" --default_suffixes "decimals =
2"...
         --counters "CPU Power (Watts) = C1 C2 0.5 * *, CPU Energy (Joules) = C1 C2 *
integral, ...
         DIMMs A Power (Watts) = C5 C6 C7 + +, DIMMs B Power (Watts) = C8 C9 C10 + +,
DIMMs...
         A + B Energy (Joules) = C5 C6 C7 + + C8 C9 C10 + + + integral, RAID Power
(Watts) =...
         C15 C16 C17 C18 + + +, RAID Energy (Joules) = C15 C16 C17 C18 + + + integral,
NIC Power...
         Energy (Joules) = C50 integral" --postfix --math_threads 4 --diagnostic --
time_integral...
         --instance_name "SUT EBI2SQC128"
      Note: ... indicates continuous lines.

   *   Third-party trademarks are the property of their respective owners.
   **  Postfix notation places the operators after the operands.  For example,
       to add 2 and 3, write 2 3 +
   *** Refer to Edsger Dijkstra's shunting-yard algorithm for an example of infix
       to postfix expressions converter.
```

2.3.1.1. Example Command Lines

Here are several (10) examples of how to start ESRV in DAQ mode:

```
esrv --start --daq --device y100 --interface_options "com=1" --device_options
"channels=20" --channels "1 2 5-10 18-15 50" --default_suffixes "decimals = 2" --
counters "Dummy=3.14,CPU Power (Watts) = C1 C2 *, CPU Energy (Joules) = C1 C2 *
integral, DIMMs A Power (Watts) = C5 C6 C7 + +, DIMMs B Power (Watts) = C8 C9 C10 + +,
DIMMs A + B Energy (Joules) = C5 C6 C7 + + C8 C9 C10 + + + integral, RAID Power
(Watts) = C15 C16 C17 C18 + + +, RAID Energy (Joules) = C15 C16 C17 C18 + + +
integral, NIC Power Energy (Joules) = C50 integral" --postfix --math_threads 4 --
diagnostic --time_integral

esrv --start --daq --library ./esrv_daq_simulated_device.so --channels "1 2 5-10 18-15
50" --default_suffixes "decimals = 2" --counters "Dummy=3.14,CPU Power (Watts) = C1 C2
*, CPU Energy (Joules) = C1 C2 * integral, DIMMs A Power (Watts) = C5 C6 C7 + +, DIMMs
B Power (Watts) = C8 C9 C10 + +, DIMMs A + B Energy (Joules) = C5 C6 C7 + + C8 C9 C10
+ + + integral, RAID Power (Watts) = C15 C16 C17 C18 + + +, RAID Energy (Joules) = C15
C16 C17 C18 + + + integral, NIC Power Energy (Joules) = C50 integral" --postfix --
diagnostic --time_integral --math_threads 4 --math_threads_priority_boost  --
kernel_priority_boost --math_threads_priority_boost

esrv --start --daq --device y100 --interface_options "com=1 bauds=115200" --
device_options "channels=20 all_channels_type=volt all_channels_range=2V" --channels
"1 10" --identity --diagnostic

esrv --start --daq --device y100 --interface_options "com=1 baud=115200" --
device_options "channels=20 all_channels_type=volt all_channels_range=2V
channel_4_range=6V channel_5_range=20V" --channels "1 10 6-4" --identity --diagnostic

esrv --start --daq --device y100 --interface_options "com=1 baud=115200" --
device_options "channels=20 all_channels_type=volt all_channels_range=2V
channel_4_range=6V channel_5_range=20V"  --channels "1 10 6-4" --default_suffixes
"decimals = 4" --counters "Power 1 (Watts) = C1 C4 +, Power 2 (Watts) = C5 C6 C10 + +,
Energy 1 (Joules) = C1 C4 + integral, Energy 2 (Joules) = C5 C6 C10 + + integral" --
diagnostic

esrv --start --daq --interface_options "com=1 baud=115200 parity=n data=8 stop=1 xon=n
dtr=y rts=n" --device y100 --channels "1-4" --default_suffixes "decimals = 2" --
counters "DIMM A (Joules) = C1 0.5 * integral, DIMM B (Joules) = C2 0.5 * integral,
DIMM C (Joules) = C3 0.5 * integral, DIMM D (Joules) = C4 0.5 * integral" --postfix --
math_threads 4 --diagnostic --time_integral

esrv --start --daq --interface_options "com=1 baud=115200 parity=n data=8 stop=1 xon=n
dtr=y rts=n" --device y100 --device_options "channels=20"  --channels "11-14" --
default_suffixes "decimals = 2" --counters "DIMM A (Joules) = C1 C1 * 2.5 * integral,
DIMM B (Joules) = C2 C2 * 2.5 * integral, DIMM C (Joules) = C3 C3 * 2.5 * integral,
DIMM D (Joules) = C4 C4 * 2.5 * integral" --postfix --math_threads 4 --diagnostic --
time_integral

esrv --start --daq --interface_options "com=1 baud=115200 parity=n data=8 stop=1 xon=n
dtr=y rts=n" --device y100 --device_options "channels=20"  --channels "11-14" --
default_suffixes "decimals = 8" --counters "DIMM A (Watts) = C1 C1 * 2.5 * , DIMM B
(Watts) = C2 C2 * 2.5 *, DIMM C (Watts) = C3 C3 * 2.5 *, DIMM D (Watts) = C4 C4 * 2.5
*" --postfix --math_threads 4 --diagnostic --time_integral
```

```
esrv --start --daq --interface_options "ip=10.23.3.27 port=34318" --device y100 --
device_options "channels=20"  --channels "11-14" --default_suffixes "decimals = 2" --
counters "DIMM A (Watts) = C1, DIMM A (Joules) = C1 integral, DIMM B (Watts) = C2,
DIMM B (Joules) = C2 integral, DIMM C (Watts) = C3, DIMM C (Joules) = C3 integral,
DIMM D (Watts) = C4, DIMM D (Joules) = C4 integral" --postfix --diagnostic --
time_integral

./esrv --start --interface_options "com=0 baud=57600" --daq --device y100 --
device_options "channels=20"  --channels "11-14" --default_suffixes "decimals = 2" --
counters "DIMM A (Watts) = C1, DIMM A (Joules) = C1 integral, DIMM B (Watts) = C2,
DIMM B (Joules) = C2 integral, DIMM C (Watts) = C3, DIMM C (Joules) = C3 integral,
DIMM D (Watts) = C4, DIMM D (Joules) = C4 integral" --postfix --time_integral --
instance_name "EBI2SQC65" --diagnostic
```

NOTE

The MW100 has both serial and Ethernet interfaces. By default, the built-in ESRV support library uses the Ethernet interface. Serial support modules are also provided in the SDK (`yokogawa_mw100s.dll` and `yokogawa_mw100s.so`). When invoking the serial interface, use the `--library` option instead of the `--device` option in the command line.

2.3.1.2. ESRV Counter Math

Because DAQs contain many channels and can be configured to measure any arbitrary parameters, it's not possible to have pre-configured, fixed counters in DAQ mode. The user must define counters based on the application and set up. These counters can be mapped one-to-one to the DAQ channels, or can be combined to a subset of the DAQ channels using a math formula (called ESRV counter equations). ESRV then applies the formula to each ESRV counter the user defines.

NOTE

A maximum of 128 ESRV counters can be defined, and 128 DAQ channels can be managed by this release of ESRV.

2.3.1.2.1. Counter Equations

Counter equations allow you to feed DAQ channel data into a defined and exposed counter using a mathematical function. Each equation uses its own LIFO stack, and data cannot be shared among counter stacks. ESRV counter equations must be expressed in *postfix* notation rather than in infix notation. Postfix takes all the operands first and all the operators second.

For example, if you want to set up a counter that reports the power draw of four DIMMs on a monitored server using four DAQ channels, you would do the following:

1. Use a DAQ channel to measure each DIMM, 1 through 4.
2. Define a counter, for example call it "Memory Subsystem Power (in watts)."
3. Sum the measurement of the four DAQ channels (power of each DIMM) into the counter output, by pushing the value of DAQ channels 1 through 4 onto the equation stack and adding them.

To define such a counter, you would use the following arguments:

`--counters "Memory Subsystem Power (in watts) = C1 C2 C3 C4 + + +".`

Note the postfix notation. Furthermore:

- The stack depth is 128.
- The data type of the stack elements is dynamic and depends on the type of the values stored in them.
- The type of the result of the evaluation of an ESRV counter equation is semi-dynamic. If only integer values and operations (as per their results) are used, then the ESRV counter value is an integer. This matches the native PL counters data type (matching the unsigned long long C data type).
- If a single floating-point argument or operational result is stored in the stack, then the ESRV counter value is a floating-point value (matches the long double C type).
- If an ESRV counter had a floating-point value, then it requires the use of suffix counters to be fully represented.
- Special instructions such as `sign`, `ip` and `fp` can be used to populate the suffix counters in ESRV counter equations (see below for details).

When adding literals, DAQ *channels* (always represented by a floating-point value, matching the long double C type) can be used in counter equations. For example, to push onto the stack the value of the DAQ channel number 5, type `C5` in your equation. To push onto the stack the value of the DAQ channel number 10, type `C10`. And so on. In the example above, the values of DAQ channels 1 to 4 are pushed onto the stack first; then, three additions are performed.

Without changing the result of this equation – but making it much slower to evaluate – you could write:

```
--counters "Memory Subsystem Power (in watts) = C1 C2 0 + C3 C4 + + + 1.0 *".
```

It is important to note that ESRV requires that each equation evaluates into a single value. If a value is left on the stack in addition to the top of stack (TOS), an error is generated. *This is a hard requirement!*

2.3.1.2.2. Types and Purposes of Counter Equations

ESRV counter equations can be composed of:

- Channel readings (`Cx` where `x` is the channel number. `C12` for channel 12, etc.)
- Literals (`1`, `-3.14`, `1E-5`, etc.)
- Operations:
    - `+`: add the two first stack levels (TOS = TOS + L1)
    - `-`: subtract the two first stack levels (TOS = TOS - L1)
    - `*`: multiply the two first stack levels (TOS = TOS * L1)
    - `/`: divide the two first stack levels (TOS = TOS / L1). An error is raised if L1 = 0
- Functions:
    - `sign`: push 1 (negative) or 0 (positive or null) to the stack according to the sign of the TOS value (TOS=sign(TOS))
    - `ip`: push the TOS value's integer part on the TOS (TOS=ip(TOS))
    - `fp`: push the TOS value's fractional part on the TOS (TOS=fp(TOS))

- `abs`: push the TOS value's absolute value on the TOS (TOS=abs(TOS))
- Stack manipulation functions:
  - `dup`: duplicate the TOS value (TOS -> TOS TOS)
  - `swap`: swap TOS and L1 (TOS L1 -> L1 TOS)
  - `drop`: remove the TOS value (TOS L1 -> L1)
- Counter qualifiers:
  - `integral`: the computed value is summed into the counter
  - `offset`: the counter can be offset

2.3.1.2.3. ESRV Equation Processing

ESRV counter equations are pre-compiled during the user input analysis at ESRV startup. During pre-compilation, each equation is converted into a stream of byte codes to speed-up computations during the measurement phase. Each ESRV counter equation can be compiled into as many as 256 op-codes. An error is raised if this value is exceeded. However, for performance considerations, equations should be as short as possible.

If the `integral` counter qualifier is found in an ESRV counter equation, then the counter's values will be integrated – or summed – over time by ESRV. The following counter definition:

```
--counters "Memory Subsystem Energy (in Joules) = C1 C2 C3 C4 + + + integral"
```

uses the same equation as the one used above. Appending the `integral` counter qualifier turns it into an *energy counter*. Indeed, *energy is the integral of power over time*. Refer to the `--time_integral` advanced option for more integration options.

The `offset` counter equation qualifier activated the offsetting of the equation value. The result of this operation is similar to the `--offset_power` and `--offset_power_samples` options of ESRV in default mode.

NOTE

`integral` *and* `offset` *ESRV counter equation qualifiers are mutually-exclusive.*

NOTE

ESRV counter equation qualifiers are not compiled into byte code and can therefore appear anywhere in the equation without syntax considerations. However, it is a good practice to put them always at the same location (at the beginning or the end of the equation).

2.3.1.2.4. Math Threads

After all the DAQ channels are read, and the device-based math computations are applied, if any, then, ESRV counter equations are evaluated. To limit the performance impact of the equation evaluations – and to leverage modern processors' multi-core architectures – ESRV distributes equation evaluations among multiple threads running in parallel. A maximum of four (4) math threads can be used in this release of ESRV.

Use the `--math_threads` option to specify the number of math threads ESRV should use, if you want less than four. ESRV tries to balance the load between math threads by following a few simple rules.

1. If there are more than eight (8) counter equations, they are split into blocks. Otherwise a single math thread is used.
2. The size of the block is equal to the total number of equations divided by the number of math threads.
3. If the block size is less than two (2), a single thread is used.
4. If the block size is greater than ten (10), a warning is issued.
5. ESRV also checks the time required by the math threads to compute the counter equations. If the computational time exceeds 40 percent of the sample interval, then a warning is issued.

NOTE

ESRV counter equations are independent from any math functions a DAQ unit may apply to its channels. Both equation types can coexist. Math functions, if performed by the device and not post-processing software, are performed prior to the channel reading by ESRV. The counter equation of ESRV applies to the channel readings as transmitted by the device.

2.3.1.3. Advanced Startup Options

ESRV offers some advanced options that can be added to the command line:

- Specify active channels (`--channels`)
- Do not apply math functions to the channels (`--identity`)
- Define counter names (`--counters`)
- Specify a file containing the counters definitions (`--counters_file`)
- Set counters default suffixes (`--default_suffixes`)
- Activate use of postfix notation (`--postfix`)
- Activate time weighting of samples (`--time_integral`)
- Set the number of math threads (`--math_threads`)
- Increase math threads' priority (`--math_threads_priority_boost`)

2.3.1.3.1. Specify active channels

DAQ units often have tens or hundreds of measurement channels. It is highly probable that a small sub-set of these channels are used simultaneously. Use the `--channels` option to identify which channels are actively used for data collection. If the DAQ ESRV support module doesn't provide an optimized channel reading function, then this option can improve ESRV's performance by limiting the number of channel reading function calls. For example, `--channels "1 2 5-10 18-15 50"` activates only channels 1, 2, 5, 6, 7, 8, 9, 10, 15, 16, 17, 18 and 50.

2.3.1.3.2. No Math Applied

To map one-to-one ESRV counters to DAQ channels, use the `--identity` option. When this option is specified, a counter is created for each DAQ channel used. In this case, each counter is named “DAQ Channel” followed by the channel count. This option cannot be used with the `--counters` option.

2.3.1.3.3. Define ESRV Counter Names

`--counters <string>` defines the ESRV counters and uses the syntax:

```
--counters "counter_name = equation, <counter_name = equation , …>"
```

The counter definitions are delimited by quotes (""); the equal sign (=) separates the counter name from the equation associated with it. Multiple counter definitions are separated by commas (,). Note that the leading and trailing spaces are trimmed when retrieving the counter name.

For example:

```
--counters "DIMMs A Power (Watts) = C5 C6 C7 + +, DIMMs B Power (Watts) = C8 C9 C10 +
+, DIMMs A + B Energy (Joules) = C5 C6 C7 + + C8 C9 C10 + + + integral"
```

defines three counters with equations.

2.3.1.3.4. Specify Counters Definitions File

ESRV counter definitions can be lengthy. To shorten ESRV run commands, you can store the counter definitions in an ASCII file and use the `--counters_file <file name>` option to point ESRV to that file. The counter definitions file is the verbatim copy of the string argument that would be used otherwise with the `--counters` option (without the leading and trailing double quotes ("")). For example, to reference the above example in a file, use the following:

```
--counters_file "three counters"
```

references the file “three counters”, which contains the definitions string:

```
DIMMs A Power (Watts) = C5 C6 C7 + +, DIMMs B Power (Watts) = C8 C9 C10 + +,
DIMMs A + B Energy (Joules) = C5 C6 C7 + + C8 C9 C10 + + + integral
```

2.3.1.3.5. Set Counters Default Suffixes

You can create and set suffix counters for ESRV counters using the `--default_suffixes <suffix_name=value>` option. You can define multiple default suffixes by separating them with spaces. Each suffix counter is defined by a suffix name (see list below), an equal sign (=) and a value.

For example, to create suffix counters for two (2) decimal digits, you would use the following:

```
--default_suffixes "decimals = 2"
```

This creates a *.decimals* counter *for each counter* and sets each one to the value 2. Default suffix counters are created and initialized at ESRV startup.

The following suffix name options are available:

- sign
- decimals
- offset
- offset.decimals

- offset.sign
- scalar
- scalar.decimals

If a suffix counter's value is not constant, then define the counter with the `--counters` option and associate it with an equation. For example, a counter can be defined as: `--counters "Counter Name.sign = C1 sign"`.

NOTE

Creating two counters with the same name is not allowed. For example, you cannot use the `--default_suffixes` *option to create a sign suffix counter, and then explicitly create a .sign suffix counter with the* `--counters` *option.*

2.3.1.3.6. Activate Postfix Notation

Channel equations in this release of ESRV must be expressed using the postfix notation. Using `--postfix` option is optional. However, future releases of ESRV may implement support for algebraic notation.

2.3.1.3.7. Activate Samples Time Weighting

If an ESRV counter equation is marked for integration (using the `integral` counter qualifier), then the value of the counter is summed over time. For example, if a counter is integrating a power measurement in watts from a DAQ channel every second, the resultant value, by definition, will be in joules (watt-seconds). However, if the ESRV sampling interval is longer, then the integration may require some adaptation. To weight the counter equation values with the sampling interval time in seconds, you would use the `--time_integral` option.

2.3.1.3.8. Set Math Threads Count

This release of ESRV can run up to four parallel (4) math threads to compute counter equations. This helps shorten computation time and to leverage multi-core processor architectures. To specify the number of threads to use, include `--math_threads <n>`, where `n` is the number of threads requested by the user.

Note that this value is a request and that ESRV may not comply. This decision is made based on a set of rules aiming to balance the computation load among threads (see section 2.3.1.2.4).

2.3.1.3.9. Increase Math Threads' Priority

Multiple threads have a maximum target execution time of 40% of the sampling interval. If ESRV is running on a heavily loaded system, the math threads may miss their target, especially when short sampling intervals are used and many equations are computed. To change thread execution priorities, use the `--math_threads_priority_boost` option. The use of this option may require higher privileges.

2.4 ESRV Integrated Device Support

ESRV has built-in support for the Yokogawa* WT210, MW100, and the Extech* 380801 external digital power readers (see Figure 8, Figure 9, and Figure 10).

Figure 8: Yokogawa* WT210 power meter

Figure 9: Yokogawa* MW100 data acquisition system

Figure 10: Extech* 380801 power meter

To use these meters, supply the appropriate device name for the selected meter:

- `--device y210` for the Yokogawa WT210 meter
- `--device e801` for the Extech 380801 meter

See the following examples.

```
esrv --start --device y210 --device_options "items=all"
esrv --start --device e801 --interface_options="com=PL2303-000103D"
```

NOTE

The Yokogawa MW100 is not a power reader. Refer to section 2.3 for details on how to use a data acquisition system with ESRV.

NOTE

The Yokogawa WT210 is only supported with a serial interface in this release of ESRV. The GPIB interface is not supported in this version of the SDK.

2.5 ESRV Library-based Device Support

In addition to the built-in support for the meters described in section 2.4, ESRV supports additional meters through the use of libraries. Library-supported devices include the following:

- Additional Yokogawa meters
- Yokogawa MW100
- APC Switched Rack power distribution unit AP7930
- ZES ZIMMER meters
- Watts up? PRO meter
- P3 International P4400 + Adafruit Industries Tweet-a-Watt Kit
- Simulated power meter
- CPU indexed simulated power meter
- Simulated data acquisition system
- Command-line interpreter
- IPMI device support (through IPMItool and the command-line interpreter)
- Custom libraries

Each of these categories is described in the following sections.

2.5.1 Additional Yokogawa Meters

In addition to built-in support for the WT210 meter, the Intel EC SDK provides libraries for three more Yokogawa meters: WT230, WT500, and WT3000 meters (see Figure 11):

Figure 11: Yokogawa WT230, WT500, and WT3000 power meters

The libraries for these meters match the following file masks:

- `yokogawa_wt*.dll` (for Windows environments)
- `yokogawa_wt*.so.1.0` (for non-Windows environments)

For example, to use the Yokogawa WT230 meter on a Windows system, use a command line like the following:

```
esrv  --start --library yokogawa_wt230.dll --interface_options="com=2 dtr=n"
--device_options "items=all"
```

Table 1 thru Table 4 summarize the device options supported by the ESRV modules.

NOTE

The Yokogawa WT210 meter is supported with a library, but there is little need to use this library since ESRV provides integrated support for that meter without needing a separate library.

Table 1: Device options for the Yokogawa WT210

Option	Value, range	Signification
`item=<string>`	A valid string among all, v1, a1, w1, va1, var1, pf1, degr1, vhz1, ahz1, wh1, whp1, whm1, ah1, ahp1, ahm1, and time	Specifies the items measured by the device. It is recommended to always request all items.
`vrange=<real>`	A valid real among 15.0, 30.0, 60.0, 150.0, 300.0, and 600.0	Specifies the voltage range to use for all channels. By default 150.0V is used.
`arange=<real>`	A valid real among 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, and 20.0	Specifies the amperage range to use for all channels. By default 5.0A is used.
`config=<string>`	A valid configuration command string	Specifies a set of configuration commands to be sent to the device during opening. Commands are separated by a ";"

		(semicolon) and must be enclosed between "'" (single quotes).
`eor=<string>`	lf, cr or crlf	Specifies the end-of-record character to be used. By default line-feed (lf) is used.

Table 2: Device options for the Yokogawa WT230

Option	Value, range	Signification
`item=<string>`	A valid string among all, f_v1, f_v2, f_v3, f_vs, f_a1, f_a2, f_a3, f_as, f_w1, f_w2, f_w3, f_ws, f_va1, f_va2, f_va3, f_vas, f_var1, f_var2, f_var3, f_vars, f_pf1, f_pf2, f_pf3, f_pfs, f_degr1, f_degr2, f_degr3, f_degrs, f_vhz1, f_vhz2, f_vhz3, f_vhzs, f_ahz1, f_ahz2, f_ahz3, f_ahzs, f_wh1, f_wh2, f_wh3, f_whs, f_whp1, f_whp2, f_whp3, f_whps, f_whm1, f_whm2, f_whm3, f_whms, f_ah1, f_ah2, f_ah3, f_ahs, f_ahp1, f_ahp2, f_ahp3, f_ahps, f_ahm1, f_ahm2, f_ahm3, f_ahms, and f_time	Specifies the items measured by the device. It is recommended to always request all items.
`vrange=<real>`	A valid real among 15.0, 30.0, 60.0, 150.0, 300.0, and 600.0	Specifies the voltage range to use for all channels. By default 150.0V is used.
`arange=<real>`	A valid real among 1.0, 2.0, 5.0, 10.0, and 20.0	Specifies the amperage range to use for all channels. By default 5.0A is used.
`config=<string>`	A valid configuration command string	Specifies a set of configuration commands to be sent to the device during opening. Commands are separated by a ";" (semicolon) and must be enclosed between "'" (single quotes).
`eor=<string>`	lf, cr or crlf	Specifies the end-of-record character to be used. By default line-feed (lf) is used.

Table 3: Device options for the Yokogawa WT500

Option	Value, range	Signification
`interface=<string>`	usb, gpib, or vxi11	Specifies the interface to be used to communicate with the unit. No default interface is selected by ESRV; you must specify an interface.
`ip=<aaa.bbb.ccc.ddd>`	A valid IP address	Specifies the unit's IP address. No default address is selected by ESRV; you must specify an address if required by the interface.
`user=<string>`	A valid string with no spaces	Specifies the user name for the login. No default user name is selected by ESRV; you must specify a valid user name if you use the unit's security features.
`password=<string>`	A valid string with no spaces	Specifies the password for the login. No default password is selected by ESRV; you must specify a valid password if you use the unit's security features.
`serial_no=<string>`	The unit's serial number as printed on the device's label	Specifies the unit to communicate with. The serial number must be typed exactly as it appears on the device's label.
`gpib_no=<integer>`	A valid integer	Specifies the GPIB port number to be used to communicate with the unit.
`item=<string>`	A valid string among all, f_v1, f_v2, f_v3, f_vs, f_a1, f_a2, f_a3, f_as, f_w1, f_w2, f_w3, f_ws, f_va1, f_va2, f_va3, f_vas, f_var1, f_var2, f_var3, f_vars, f_pf1, f_pf2, f_pf3, f_pfs, f_degr1, f_degr2, f_degr3, f_degrs, f_vhz1, f_vhz2, f_vhz3, f_vhzs, f_ahz1, f_ahz2, f_ahz3, f_ahzs, f_wh1, f_wh2, f_wh3, f_whs, f_whp1, f_whp2, f_whp3, f_whps, f_whm1, f_whm2, f_whm3, f_whms, f_ah1, f_ah2, f_ah3, f_ahs, f_ahp1, f_ahp2, f_ahp3, f_ahps, f_ahm1, f_ahm2, f_ahm3, f_ahms, and f_time	Specifies the items measured by the device. It is recommended to always request all items.
`vrange=<real>`	A valid real among 15.0, 30.0, 60.0,	Specifies the voltage range to use for all

	150.0, 300.0, 600.0, and 1000.0	channels. By default 150.0V is used.
arange=<real>	A valid real among 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, and 40.0	Specifies the amperage range to use for all channels. By default 5.0A is used.
config=<string>	A valid configuration command string	Specifies a set of configuration commands to be sent to the device during opening. Commands are separated by a ";" (semicolon) and must be enclosed between" '" (single quotes).
eor=<string>	lf, cr, or crlf	Specify the end-of-record character to be used. By default line-feed (lf) is used.

The Yokogawa WT500 accepts the following device options strings:

```
interface = <gpib | vxi11 | usb>
```

The GPIB interface requires a gpib_no= option. The vxi11 interface requires at least an ip= option (ip address in the aaa.bbb.ccc.ddd format) and, depending on the WT500 settings, it may require a user= and password= option.

The USB interface requires a serial_no= option (serial number of the WT500).

Below is an example ESRV start command using the WT500 through the USB interface:

```
esrv --start --library yokogawa_wt500.dll --device_options "interface=usb
serial_no=91H722925 items=all" --diagnostic
```

Table 4: Device options for the Yokogawa WT3000

Option	**Value, range**	**Signification**
interface=<string>	usb, gpib, or vxi11	Specifies the interface to be used to communicate with the unit. No default interface is selected by ESRV; you must specify an interface.
ip=<aaa.bbb.ccc.ddd>	A valid IP address	Specifies the unit's IP address. No default address is selected by ESRV; you must specify an address if required by the interface.
user=<string>	A valid string with no spaces	Specifies the user name for the login. No default user name is selected by ESRV; you must specify a valid user name if you use the unit's security features.
password=<string>	A valid string with no spaces	Specifies the password for the login. No default password is selected by ESRV; you must specify a valid password if you use the unit's security features.
usb_no=<integer>	A valid integer	Specifies the USB port number to be used to communicate with the unit.

`gpib_no=<integer>`	A valid integer	Specifies the GPIB port number to be used to communicate with the unit.
`item=<string>`	A valid string among all, f_v1, f_v2, f_v3, f_vs, f_a1, f_a2, f_a3, f_as, f_w1, f_w2, f_w3, f_ws, f_va1, f_va2, f_va3, f_vas, f_var1, f_var2, f_var3, f_vars, f_pf1, f_pf2, f_pf3, f_pfs, f_degr1, f_degr2, f_degr3, f_degrs, f_vhz1, f_vhz2, f_vhz3, f_vhzs, f_ahz1, f_ahz2, f_ahz3, f_ahzs, f_wh1, f_wh2, f_wh3, f_whs, f_whp1, f_whp2, f_whp3, f_whps, f_whm1, f_whm2, f_whm3, f_whms, f_ah1, f_ah2, f_ah3, f_ahs, f_ahp1, f_ahp2, f_ahp3, f_ahps, f_ahm1, f_ahm2, f_ahm3, f_ahms, and f_time	Specifies the items measured by the device. It is recommended to always request all items.
`vrange=<real>`	A valid real among 15.0, 30.0, 60.0, 100.0, 150.0, 300.0, 600.0, and 1000.0	Specifies the voltage range to use for all channels. By default 150.0V is used.
`arange=<string>`	A valid string among 5mA, 10mA, 20mA, 50mA, 100mA, 200mA, 500mA with a 2A input module; or among 1A, 2A, 500mA, 1A, 2A, 5A, 10A, 20A, or 30A with 30A input module.	Specifies the amperage range to use for all channels. By default 5.0A is used.
`input_module_amperage=<real>`	2.0 or 30.0	Specifies the amperage of the input module used. This option influences the current measurement ranges available.
`config=<string>`	A valid configuration command string	Specifies a set of configuration commands to be sent to the device during opening. Commands are separated by a ";" (semicolon) and must be enclosed between "'" (single quotes).

`eor=<string>`	lf, cr, or crlf	Specify the end-of-record character to be used. By default line-feed (lf) is used.

All Yokogawa power readers accept an *`items`* option in the `device_options` string, and the *`items=all`* string should always be used with Yokogawa meters.

The list below shows all the items accessible for a three-channel device such as the WT230, WT500 and WT3000 (the WT210 recognizes only items suffixed with a '1'):

v1, v2, v3, vs, a1, a2, a3, as, w1, w2, w3, ws, va1, va2, va3, vas, var1, var2, var3, vars, pf1, pf2, pf3, pfs, degr1, degr2, degr3, degrs, vhz1, vhz2, vhz3, vhzs, ahz1, ahz2, ahz3, ahzs, w1, w2, w3, ws, whp1, whp2, whp3, whps, whm1, whm2, whm3, whms, ah1, ah2, ah3, ahs, ahp1, ahp2, ahp3, ahps, ahm1, ahm2, ahm3, ahms and *time*

ESRV uses a subset of the information returned by the meters to provide the counters described in section 2.6.

NOTE

The configuration option (`config=<string>`*) allows a user to pass device-specific-configuration commands that are not supported through device options of an ESRV support module. This is a work around to activate and/or de-activate device settings that are not exposed by ESRV. The commands are executed as-is and in the specified order. The commands are executed at the end of the initialization of the device by the ESRV support module. As a last resort, this option also allows overwriting settings requested via the device options string.*

2.5.2 Yokogawa MW100 DAQ

The Intel EC SDK provides built-in support and support libraries for the Yokogawa MW100 DAQ (see Figure 12):

Figure 12: Yokogawa* MW100 data acquisition system

The *built-in support* communicates with the MW100 using the *ethernet interface*, while the *support module* communicates with the MW100 using the *serial interface*. The libraries for this DAQ using the serial interface include:

- yokogaw_mw100s.dll (for Windows environments)
- yokogaw_mw100s.so.1.0 (for non-Windows environments)

For example, to use the Yokogawa MW100 on a Windows system, use a command line like the following:

```
esrv --start --daq --device y100 --interface_options="ip=10.23.3.27 port=34318"  --
device_options "eor = lf channels = 20  all_channels_type = volt all_channels_range =
6v channel_2_type = pulse channel_2_range = contact"
```

or

```
esrv --start --daq --library yokogawa_mw100s.dll --interface_options="com=5
baud=19200"  --device_options "eor = lf channels = 20  all_channels_type = volt
all_channels_range = 6v channel_2_type = pulse channel_2_range = contact"
```

Table 5 summarizes device options for the MW100.

Table 5: Device options for the Yokogawa MW100

Option	**Value, range**	**Signification**
channels=<integer>	A valid integer between 1 and 60	Specifies the number of channels available in the MW100 configuration. The MW100 is a modular unit and can have a variable number of channels. This number ***must*** be equal to the total number of channels ***available***; it is ***not*** the number of channels used. No default channel count is selected; you must specify a channel count.
all_channels_type=<string>	A valid string among volt, tc, rtd, di, ohm, str, or pulse	Specifies the type of all channels.
channel_x_type=<string>		Specifies the type of channel x.
all_channels_range=<string>	See Table 6	Specifies the range of all channels.
channel_x_range=<string>		Specifies the range of channel x.
eor=<string>	lf, cr, or crlf	Specifies the end-of-record character to be used. By default line-feed (lf) is used.

Table 6: Range options available for each channel type

Type	**Value, range**
Volt	20mv, 60mv, 200mv, 2v, 6v, 20v, 100v, 60mvh, 1v, or 6vh
Tc	rsbkejtnwlu, kpvsau7fe, platinel, pr40-20, ninimo, wre3-25, wwre26, n14, or xk
Rtd	pt100-1, pt100-2, jpt100-1, jpt100-2, pt100-1h, pt100-2h, jpt100-1h, jpt100-2h, ni100sama, ni100din, ni120, pt50, cu10ge, cu10ln, cu10weed, cu10bailey, j263b, cu10a392, cu10a393, cu25, cu53, cu100, pt25, cu10geh, cu10lnh, cu10weedh, cu10baileyh, pt100-1r, pt100-2r, jpt100-1r, jpt100-2r, pt100g, cu100g, cu50g, cu10g, pt500, or pt1000
Di	Level, or contact

Ohm	20ohm, 200ohm, ro 2000ohm
Str	2000ustr, 20000ustr, or 200000ustr
Pulse	Level, or contact

NOTE

`channel_x_type=` *and* `channel_x_range=` *options can be used to overwrite a general channel setting (*`all_channels_type=` *and* `all_channels_range=`*) for a given channel (*`x`*). These settings must be specified* ***after*** *the general setting. Otherwise, they will be ignored and only the general settings will be used.*

NOTE

Refer to your unit documentation for details on these options and values for valid ranges if not specified in this guide.

2.5.3 APC Switched Rack Power Distribution Unit AP7930

The Intel EC SDK provides libraries for the AP7930 switched rack power distribution unit (see Figure 13):

Figure 13: APC Switched Rack Power Distribution Unit AP7930

The libraries for this power distribution unit are:

- `apc_switched_rack_pdu.dll` (for Windows environments)
- `apc_switched_rack_pdu.so.1.0` (for non-Windows environments)

For example, to use the APC Switched Rack PDU AP7930 on a Windows system, use a command line like the following:

```
esrv  --start --library apc_switched_rack_pdu.dll
--interface_options="ip=10.23.30.40 port=23 user=administrator password=Zx#&!178"
```

NOTE

This support library does not have options. All the options must be provided via the command line option `--interface_options` *of ESRV. The communication between ESRV and the PDU is done* through the Ethernet interface using the Telnet protocol.

Table 7 summarizes these options.

Table 7: Interface options for the APC Switched Rack PDU

Option	Value, range	Signification
ip=<aaa.bbb.ccc.ddd>	A valid IP address	Specifies the unit's IP address. By default ESRV uses 127.0.0.1.
port=<integer>	A valid port number	Specifies the TCP port. By default ESRV uses 27015. Port 23 is expected by the unit.
user=<string>	A valid string with no spaces	Specifies the user name for the login. By default ESRV uses 'user.' 'apc' is the default user name of the unit
password=<string>	A valid string with no spaces	Specifies the password for the login. By default ESRV uses 'password.' 'apc' is the default password of the unit

NOTE

The default Ethernet interface options are generic to all devices supported by ESRV and using the Ethernet interface. This is why the default port is not 23 (as Telnet expects it) but 27015, and so on.

NOTE

The integrated CLI of the PDU can be accessed by appending `-c` *to the password. The support module does this automatically, so you do not have to provide this extension in the password option.*

NOTE

The unit may have a time-out period set. If the `--pause` *option of ESRV is set to specify a pause time longer than the time-out period, then the connection may be broken by the PDU, and the module will fail to communicate with the unit afterward. Either de-activate the PDU timeout feature or shorten the ESRV sampling interval to communicate with the PDU and keep the connection alive.*

2.5.4 ZES ZIMMER Meters

The Intel EC SDK provides libraries for three ZES ZIMMER meters: LMG95, LMG450, and LMG500 meters (see Figure 14):

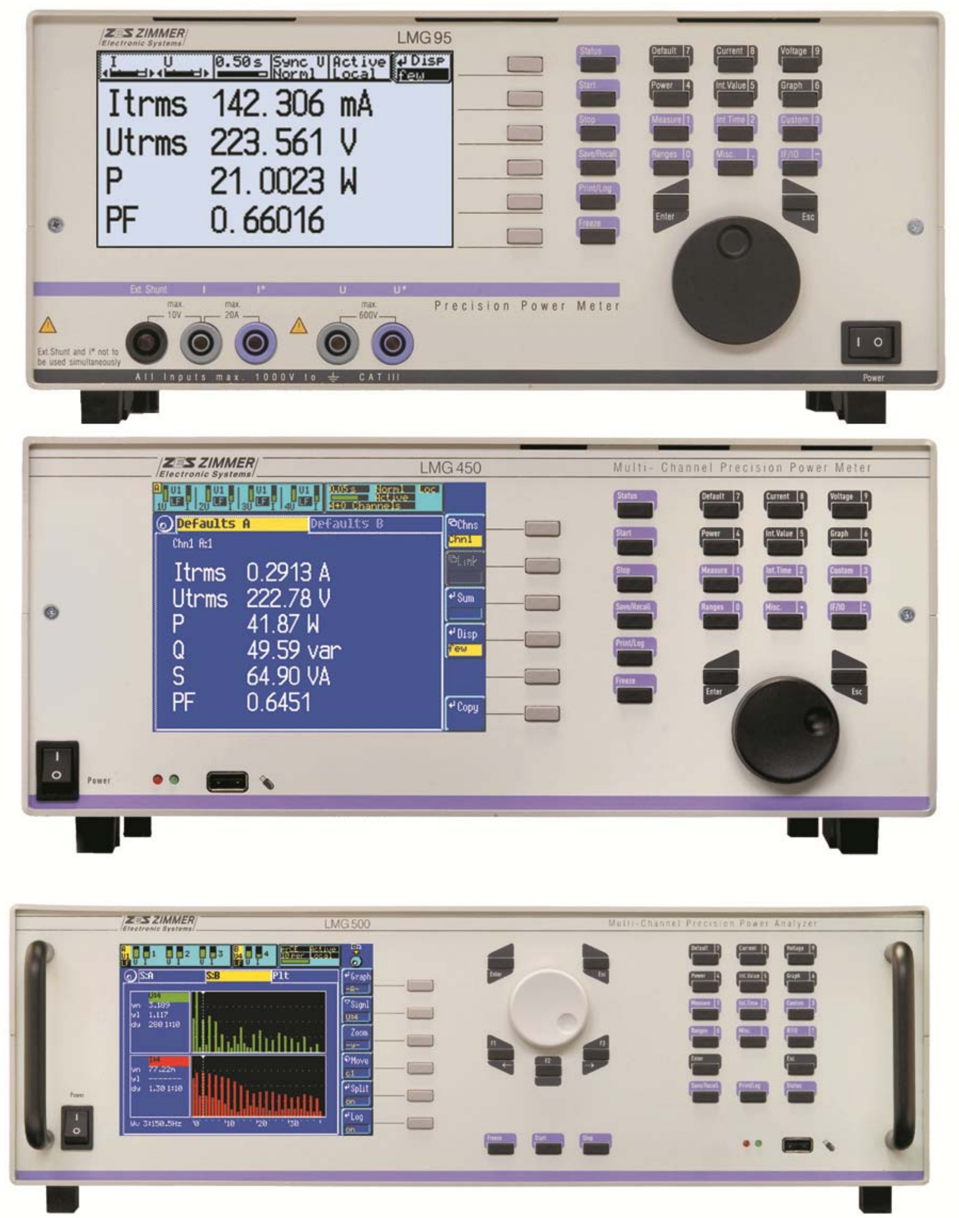

Figure 14: ZES ZIMMER LMG95, LMG450, and LMG500 power meters

The libraries for these meters match the following file masks:

`zes_zimmer_lmg*.dll` (for Windows environments)

`zes_zimmer_lmg*.so.1.0` (for non-Windows environments).

For example, to use the ZES ZIMMER LM450 meter on a Windows system, use a command line like the following:

```
esrv  --start --library zes_zimmer_lmg450.dll --interface_options="com=2 "
--device_options "channels=1&3 eor=cr arange=16 vrange=600"
```

Table 8, Table 9, and Table 10 summarize the device options supported by the ESRV module.

Table 8: Device options for the ZES ZIMMER LMG95

Option	**Value, range**	**Signification**
vrange=<real>	A valid real	Specifies the voltage range to use for all channels. By default 130.0V is used.
arange=<real>	A valid real	Specifies the amperage range to use for all channels. By default 5.0A is used.
config=<string>	A valid configuration command string	Specifies a set of configuration commands to be sent to the device during opening. Commands are separated by a “;” (semicolon) and must be enclosed between “'” (single quotes). No ;*OPC? should be added, and no ;*OPC? will be added.
eor=<string>	lf, cr, or crlf	Specifies the end-of-record character to be used. By default line-feed (lf) is used.

Table 9: Device options for the ZES ZIMMER LMG450

Option	**Value, range**	**Signification**
channels=<integer>[&<integer>]	Integer between 1 and 4	Specifies the channels to read. By default the first channel is read. If multiple channels must be read, specify their numbers by inserting an “&” character. For example, to read channels 1 and 3, enter “1&3” with no spaces around the “&”.
vrange=<real>	A valid real	Specifies the voltage range to use for all channels. By default 130.0V is used.
arange=<real>	A valid real	Specifies the amperage range to use for all channels. By default 5.0A is used.
config=<string>	A valid configuration command string	Specifies a set of configuration commands to be sent to the device during opening. Commands are separated by a “;” (semicolon) and must be enclosed between “'” (single quotes). No ;*OPC? should be added, and no ;*OPC? will be added.
eor=<string>	lf, cr, or crlf	Specifies the end-of-record character to be used. By default line-feed (lf) is used.

Table 10: Device options for the ZES ZIMMER LMG500

Option	Value, range	Signification
channels=<integer>[&<integer>]	Integer between 1 and 8	Specifies the channels to read. By default the first channel is read. If multiple channels must be read, specify their number by inserting an "&" character. For example, to read channels 1 and 7, enter "1&7" with no spaces around the "&".
vrange=<real>	A valid real	Specifies the voltage range to use for all channels. By default 130.0V is used.
arange=<real>	A valid real	Specifies the amperage range to use for all channels. By default 5.0A is used.
config=<string>	A valid configuration command string	Specifies a set of configuration commands to be sent to the device during opening. Commands are separated by a ";" (semicolon) and must be enclosed between "'" (single quotes). No ;*OPC? should be added, and no ;*OPC? will be added.
eor=<string>	lf, cr, or crlf	Specify the end-of-record character to be used. By default line-feed (lf) is used.

NOTE

Refer to your unit documentation for details on these options and values valid ranges if not specified in this guide.

NOTE

The configuration option allows a user to pass device-specific configuration commands that are not supported through device options. This is a work around to activate / de-activate device settings that are not exposed by ESRV. The commands are executed as-is and in the specified order. The commands are executed at the end of the initialization of the device by the ESRV support module. This also allows overwriting options set via the device options string.

2.5.5 Watts up? PRO Meter

The Intel EC SDK provides libraries for the Watts up? PRO meters (see Figure 15):

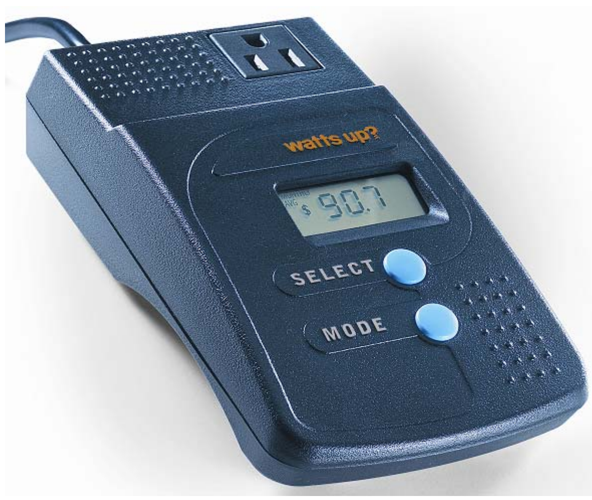

Figure 15: Watts up? PRO power meter

The libraries for this meter are:

- `watts_up_pro.dll` (for Windows environments)
- `watts_up_pro*.so.1.0` (for non-Windows environments).

For example, to use the Watts up? PRO meter on a Windows system, use a command line like the following:

```
esrv  --start --library watts_up_pro.dll --interface_options="com=0 baud=115200" --
device_options "harware_integration"
```

To use the Watts up? PRO meter on a Linux system, use a command line like the following:

```
esrv  --start --library watts_up_pro.dll --interface_options="com=0 baud=115200
mode=usb" --device_options "harware_integration"
```

where `0` is the ttyUSB# assigned to the meter.

Table 11 summarizes this option.

Table 11: Device option for the Watts up? PRO meter

Option	Value, range	Signification
`hardware_integration`	None	Activates the use of the unit's power integration. It is recommended to use this device option.

NOTE

The Watts up? PRO meter has a USB interface. To use this meter with ESRV, your operating system needs to support serial over USB functionality, usually via a driver. Solaris 10 does not have a driver; it is not supported.

2.5.6 P3 International P4400 + Adafruit Industries Tweet-a-Watt Kit

The Intel EC SDK was used for a project to convert a P3 International P4400 power meter and an Adafruit Industries Tweet-a-Watt kit into a wireless power meter solution (see Figure 16).

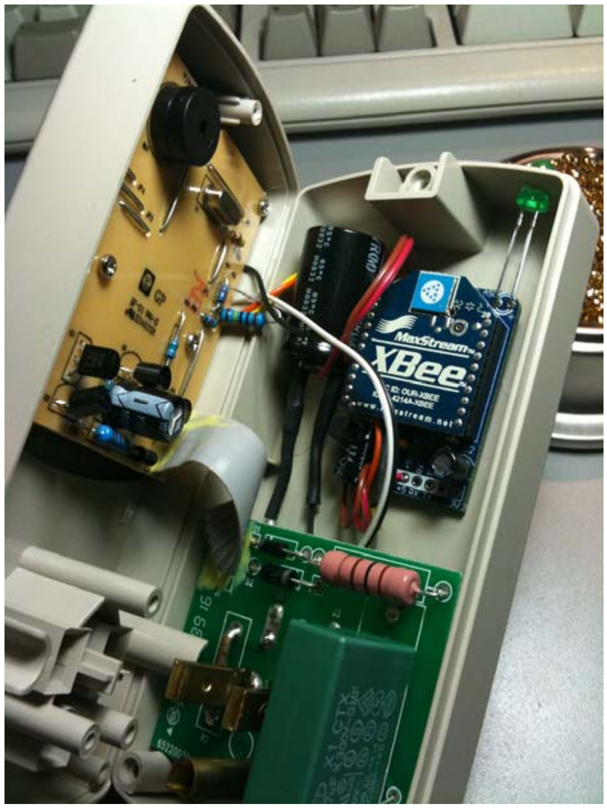

Figure 16: P3 International P4400 power meter and an Adafruit Industries Tweet-a-Watt kit assembled (shown opened)

A blog entry describing the building of the kit and the ESRV support library code can be found at the following web address:

http://software.intel.com/en-us/blogs/2010/04/15/using-the-intel-energy-checker-sdk-at-home/

The device libraries are described on the blog. For example, to use the kit on a Windows system, use a command line like the following:

```
esrv  --start --library p3_kill_a_watt_adafruit.dll --interface_options="com=9"
--device_options "channels=3 offsets=500,498,502 aggregate_channels"
```

Table 12 summarizes these options.

Table 12: Device options for the Tweet-a-Watt kit

Option	Value, range	Signification
`channels=<integer>`	Integer between 1 and 23	Specifies the number of Tweet-a-Watt Kits participating in the Personal Area Network (PAN). This value is 1 by default. Note: each unit is considered by ESRV as a measurement channel, hence the name of the option.
`aggregate_channels`	None	Specifies that all the channel readings are aggregated into a single channel (channel 1).
`calibrate`	None	Specifies that an automatic calibration phase has to be performed. Note that during calibration, no load should be applied to the unit(s).
`calibrate_samples=<integer>`	Integer between 1 and 1024	Specifies the number of samples to take during the calibration phase. By default, 10 samples are taken.
`offsets=<o1>,...,[<on>]`	Vector of integers. Each integer must be comprise between 0 and 1023 (inclusive)	Specifies for each unit the DC offset to use. There must be as many offset values as units. Note that this option and the calibrate options are mutually exclusive.

2.5.7 Simulated Power Meter

Application developers may want to test or demonstrate their application integration when an external meter or instrumented power supply is not available. The Intel® EC SDK provides a simulated power meter that emulates a power draw of approximately 150 watts. The C programming notation below shows the formulas used in this simulated meter:

```
power = 150.0 + ((double)(rand() % 10) / 10.0);
current = 5.0 + ((double)(rand() % 10) / 10.0);
voltage = 110.0 + ((double)(rand() % 10) / 10.0);
power_factor = 0.0 + ((double)(rand() % 10) / 10.0);
voltage_frequency = 50.0 + ((double)(rand() % 10) / 10.0);
current_frequency = 50.0 + ((double)(rand() % 10) / 10.0);
```

To use this simulated meter, use one of the following command lines:

```
esrv --start --library esrv_simulated_device.dll
./esrv --start --library ./esrv_simulated_device.so.1.0
```

2.5.8 CPU Indexed Simulated Power Meter

As a variation on the ESRV simulated device, developers can use a CPU-indexed ESRV simulated device. The support libraries offer several options to adapt the value of the simulated power reading using the CPU utilization percentage as a basis. The ESRV support libraries for this variation are:

- `esrv_cpu_indexed_simulated_device.dll` (for Windows environments)
- `esrv_cpu_indexed_simulated_device.so.1.0` (for non-Windows environments)

For example, to simulate a fixed power draw of 100 watts plus a variable power draw of 51.2 watts, of which 10.99 percent is indexed to the CPU utilization percentage, use a command line like the following (for a Windows environment):

```
esrv  --start --library esrv_simulated_device.dll --device_options "fixed_power=100
variable_power=51.2 cpu_power_percentatage=10.99"
```

Table 13 summarizes these options.

Table 13: Device options for the CPU indexed simulated device

Option	Value, range	Signification
`fixed_power=<real>`	A real value	Specifies the fixed power draw in watts for the simulation. By default, this value is 60.0 watts.
`variable_ power=<real>`	A read value	Specifies the variable power draw in watts for the simulation. By default, this value is 40.0 watts.
`cpu_power_percentage=<real>`	A valid string with no spaces	Specify the percentage of the variable power draw in watts for the simulation. By default, this value is 25.0 percent. This means that if the CPU utilization is 100 percent, then 25 percent of the variable power draw is added to the fixed power draw.

NOTE

The source code of the CPU indexed simulated device is supplied in the SDK Device Kit so it can be studied and expanded to use different system-level metrics to simulate the power draw.

2.5.9 Simulated Data Acquisition System

Application developers may want to test or demonstrate their application integration when an external DAQ is not available. The Intel EC SDK provides a simulated DAQ with ten (10) simulated channels that emulates a power draw of approximately 5 watts per channel. The C programming notation below shows the formulas used in this simulated meter:

```
p->daq_channel_readings[p->daq_active_channels[c]] =
   5.0 +
   (
      (double)(
         rand() %
         10
      )
      /
      10.0
   )
;
```

To use this simulated meter, use one of the following command lines:

```
esrv --start --daq --library esrv_daq_simulated_device.dll
esrv --start --daq --library ./esrv_daq_simulated_device.so
```

2.5.10 ESRV Command-line Library Support

The Intel Energy Checker SDK is shipped with an ESRV library that parses output from external applications to read device power, energy consumption, and other data via ESRV. In particular, this can be used with IPMItool to query and report appropriate IPMI sensor values as described in section 2.5.11. This command line support can work with virtually any application that prints power information to the standard output. In the examples below, a fictitious `my_utility` application is often used for illustration.

2.5.10.1. Notation Used in Command-line Library

The notation used to describe the command line options often involves the inclusion of a command line parameter. The following data types are used:

`<integer>` is an integer as defined by `[whitespace] [+] [digits]`

`<double>` is a double floating-point value as defined by

`[whitespace] [sign] [digits] [.digits] [ {d | D | e | E }[sign]digits]`.

`<string>` is an ASCII string containing only shell-authorized characters. If the string has spaces, then it must be enclosed in quotes (“”).

NOTE

To quote strings inside of other strings, the following approach should be used: the top level uses double quotes ("), the next level uses single quotes ('), and the third level uses a backslash in front of double quotes (\"). Strings used within the double quotes (") delimiting the `--device_options` *argument should use single quotes (').*

2.5.10.2. ESRV Command-line Library Syntax

The ESRV command-line library issues commands to the external tool and reads the tool's output. If required, the library performs an additional parsing step of the output to isolate the data. Input and output to the tool must be in ASCII format.

The general format for using the ESRV command-line library is as follows (this example calls the library in a non-Windows environment):

```
esrv --start --library ./command_line.so.1.0 --device_options <string>
```

The <string> argument includes multiple embedded elements to perform the following actions:

- Specify the tool to use (--tool, --shell)
- Initialize and close the tool (--open_command, --close_command)
- Set the sampling interval (--sampling_interval)
- Characterize the device (--hardware_integration, --energy_overflow)
- Manage energy integration (--read_energy_*_command)
- Define tokens (--read_*_separators, --read_*_previous_token)
- Read energy and power (--read_*_command)

The <string> argument must always be in double quotes ("") since multiple elements must be specified for the ESRV command-line library.

The following sections provide details about each of these options.

2.5.10.2.1. Specify the Tool to Use

This option is mandatory.

Use the --tool <string> parameter to identify the application the ESRV command-line library will use to gather power data. The argument must be either a fully qualified executable filename or the name of an executable found in the path environment variable. For example, to run my_utility from the /bin directory, include the following within the device_options argument:

```
--tool /bin/my_utility
```

By default, the selected tool will be invoked each time ESRV reads power or energy data. Some tools (i.e., IPMItool) support shell modes. In these modes, the tool only needs to be invoked once, and commands can be repeatedly sent to the same instance of that tool (see section 2.5.10.2.8 for a discussion of performance implications).

To enable shell mode, use the optional --shell flag. For example, to run my_utility in shell mode, include the following in the device_options argument:

```
--tool my_utility --shell
```

NOTE

The --shell flag tells the ESRV command-line library to operate in shell mode; it does nothing specifically to put the external tool in shell mode. If a command-line option is needed to start the tool in shell mode, that option should be specified in the argument provided in the --open_command string (see section 2.5.10.2.2 next).

2.5.10.2.2. Initialize and Close the Tool

Use the optional --open_command <string> parameter to identify any commands to be sent to the tool as command-line parameters when it is started. If the --shell flag is used, --open_command will only be used once; otherwise, it will be used each time ESRV collects data.

For example, to run my_utility and initialize it with a "quiet mode" command string, include the following within the device_options argument:

```
--tool my_utility --open_command 'quiet mode'
```

NOTE

Since the "quiet mode" string has spaces, it must be quoted. Since it will be placed within the device_options argument, which uses double quotes (""), this command must use single quotes ('') to enclose a string.

Similarly, use the optional --close_command <string> parameter to do any application cleanup before exiting. For example, if the tool needs a bye command to exit, the device_options string might include the following:

```
--tool my_utility --close_command bye
```

NOTE

During system integration and debugging, incorrect parameter exchanges between the ESRV command-line library and the external tool may require the user to manually terminate the external tool each time until the proper shutdown sequence is determined. If the external tool does not terminate properly, ESRV may not terminate.

2.5.10.2.3. Set the Sampling Interval

By default, ESRV will call the data sampling routines at one-second intervals (or at whatever time interval is specified with the --pause command line parameter to ESRV). In rare cases, it may be necessary to increase the interval between samples via the ESRV command-line library.

To increase the interval between samples, set the optional --sampling_interval <integer> to a value greater than one. For example, to only sample power with my_utiity at one-third the normal ESRV rate, include the following within the device_options argument:

```
--tool my_utility --sampling_interval 3
```

NOTE

*The sampling_interval is a **multiplier** of the pause parameter. If sampling_interval is set to 15 and pause is set to 20, ESRV will generate data every 20 seconds but the data will be repeated 15 times. To avoid false interpretations of data, it is generally advisable to omit the command-line library's sampling_interval and set the ESRV pause parameter to the total desired interval between samples.*

2.5.10.2.4. Characterize the Device

ESRV can *estimate* energy consumption through software integration of power readings (energy ≈ sampled_power * interval_between_samples). Meters or power supplies that perform hardware integration, however, can more accurately characterize energy consumption by sampling power much more frequently. [Some power meters internally sample power more than 100,000 times per second, even if they are only reporting out the average power each second.] If a device provides hardware integration, the ESRV command-line library can query the device for energy consumption statistics if the proper parameters are supplied.

By default, the ESRV command-line library assumes that a device does not support hardware integration, and ESRV must perform software integration of energy. To indicate that the device does support energy integration in the hardware, provide the optional `--hardware_integration` flag.

For example, to indicate that `my_utiity` can provide energy consumption statistics directly, include the following within the `device_options` argument:

```
--tool my_utility --hardware_integration
```

As noted in section 2.2.4, the 64-bit counter values used by ESRV are unlikely to overflow. However, some power meters with hardware integration record energy consumption in a manner that reaches its maximum value sooner than a 64-bit counter. For example, the Yokogawa power readers overflow at 999,999 MWh (~3.6E+15 Joules), a very large number.

If there is any likelihood of overflowing a device counter, the optional `--energy_overflow <double>` parameter can be specified to identify a sentinel value that is nearing the upper limit for the device.

For example, to indicate a sentinel value for the Yokogawa meters, include the following within the `device_options` argument:

```
--tool my_utility --hardware_integration --energy_overflow 3.3E+15
```

When ESRV reads an energy value in excess of this sentinel value, it will reset the device's energy counter and increment the associated `Energy Overflows (no unit)` counter. See section 2.5.10.2.5 for details on managing energy integration.

2.5.10.2.5. Manage Energy Integration

If the `--hardware_integration` flag is set, the ESRV command-line library can issue certain commands to the power meter (or other instrumented device) to initialize, start, reset, and stop the energy integration function in the meter. Each of these optional parameters is shown below:

- Initialize the energy integration (`--read_energy_init_command <string>`)
- Start the energy integration (`--read_energy_start_command <string>`)
- Reset the energy integration (`--read_energy_reset_command <string>`)
- Stop the energy integration (`--read_energy_stop_command <string>`)

For example, to issue a `Reset kWh` command whenever `foo` needs to reset its energy accumulation counter, include the following within the `device_options` argument:

```
--tool foo --hardware_integration --read_energy_reset_command 'Reset kWh'
```

2.5.10.2.6. Define Tokens

The ESRV command-line library issues commands to the external tool and reads the tool's output. Unfortunately, the output from the tool is usually accompanied by some surrounding text and is rarely just a number that can be directly interpreted. For the command-line library to understand how to parse the tool's output, it needs to understand how to delimit tokens in the output stream and how to identify what token in the output stream precedes the value of interest. To support this tokenizing, the ESRV command-line library supports two options for reading the power parameters:

```
--read_power_separators <string>
--read_power_previous_token <string>
```

For example, consider the following output from a command line utility used to read power from a power supply. The information of interest in this example is the value '130' preceded by the word 'Reading'.

```
Sensor ID              : Avg Power (0x2e)
Entity ID              : 21.0
Sensor Type (Analog)   : Current
Sensor Reading         : 130 (+/- 0) Watts
Status                 : ok
Lower Non-Recoverable : 1920.000
Lower Critical         : na
Lower Non-Critical     : 1920.000
Upper Non-Critical     : 0.000
Upper Critical         : na
Upper Non-Recoverable : na
Assertion Events       :
Assertions Enabled     :
```

In this example, colons (:) and spaces are used to delimit tokens. If the above example was from device `foo`, to delimit and identify the values of interest (`130` and `Reading`), you would include the following within the `device_options` argument:

```
--tool foo --read_power_separators ': ' --read_power_previous_token Reading
```

Since the response to different commands may be structured differently, ESRV supports the following *separator values* defined in section 2.5.10.2.7:

- Read current: `--read_current_separators <string>`
- Read voltage: `--read_voltage_separators <string>`
- Read power factor: `--read_power_factor_separators <string>`
- Read voltage frequency: `--read_voltage_frequency_separators <string>`
- Read current frequency: `--read_current_frequency_separators <string>`
- Read energy: `--read_energy_separators <string>`

Similarly, the ESRV command-line library supports the following *predecessor tokens* defined in section 2.5.10.2.7:

- Read current: `--read_current_previous_token <string>`
- Read voltage: `--read_voltage_previous_token <string>`
- Read power factor: `--read_power_factor_previous_token <string>`
- Read voltage frequency: `--read_voltage_frequency_previous_token <string>`

- Read current frequency: `--read_current_frequency_previous_token` <string>

NOTE

The ESRV command-line library does not handle escape codes to specify token separators.

2.5.10.2.7. Read Energy and Power

The ESRV command-line library issues commands to the external tool to retrieve power, energy, and other statistics from the meter or instrumented device. For maximum flexibility, each of these commands is independently specified:

- Read power: `--read_power_command` <string>
- Read current: `--read_current_command` <string>
- Read voltage: `--read_voltage_command` <string>
- Read power factor: `--read_power_factor_command` <string>
- Read voltage frequency: `--read_voltage_frequency_command` <string>
- Read current frequency: `--read_current_frequency_command` <string>
- Read energy: `--read_energy_command` <string>

For example, if `my_utility` needs the `Retrieve Power` command to determine the current power draw, include the following within the `device_options` argument:

```
--tool my_utility --read_power_command 'Retrieve Power'
```

The `--read_power_command` parameter is ***mandatory***, but the other parameters are *optional*; however, the `--read_energy_command` parameter ***becomes mandatory*** when the `--hardware_integration` parameter is specified.

Each of the parameters defining commands to read energy and power usually requires corresponding parameters to tokenize the output. For example, if a command line includes the `--read_energy_command` parameter, it should also include the following parameters:

- `--read_energy_separators`
- `--read_energy_previous_token`

2.5.10.2.8. Performance Considerations

Using a command-line tool to query a power reading device may introduce unwanted performance penalties. For example, each time IPMItool (see section 2.5.11) is invoked, a connection to the Baseboard Management Controller (BMC) is made.

This operation introduces a non-negligible processing overhead, which makes the use of IPMItool impractical at the default ESRV sampling interval of one second between samples. One option could be to increase the pause between ESRV samples (set using the `--pause` ESRV option), but this reduces the amount of available data.

An alternate approach that minimizes performance overhead is to use the `--shell` option whenever possible. This requires that the tool offers a shell-like interface but offers substantial performance benefits.

IPMItool is one example utility that offers a shell interface if the command line for IPMItool includes the command line parameter `shell`. By specifying the `--shell` option in the `device_options` string for the ESRV command-line library, the ESRV library will start in its shell mode and keep the hosting process alive until ESRV stops. This way, each item query will be performed without the initialization overhead required to start IPMItool each time. The shell mode should be used if available.

2.5.11 ESRV IPMI Device Support

Many servers manage their power, temperature, and other platform sensors via IPMI using IPMItool (http://ipmitool.sourceforge.net/). The ESRV command-line library can use IPMItool to query on-board sensors, including sensors for IPMI-instrumented power supplies. For example, an IBM* System x3550 M2 server has a sensor called `Avg Power`.

To query this sensor on a system running Linux using IPMItool and the ESRV command-line library, use the following ESRV command line:

```
./esrv --start --library ./command_line.so.1.0 --device_options "--shell --tool
/usr/bin/ipmitool --open_command shell --read_power_command 'sensor get \"Avg Power\"'
--read_power_separators ' :' --read_power_previous_token Reading" –diagnostic
```

Note the `read_power_command` command in the example above:

`--read_power_command 'sensor get \"Avg Power\"'`

This is equivalent to the `sensor get "Avg Power"` command in IPMItool.

NOTE

IPMI sensor names can vary but the format for reading the sensors will be similar to the example above.

The output of the IPMItool command outlined above is shown below:

```
Sensor ID              : Avg Power (0x2e)
Entity ID             : 21.0
Sensor Type (Analog)  : Current
Sensor Reading        : 130 (+/- 0) Watts
Status                : ok
Lower Non-Recoverable : 1920.000
Lower Critical        : na
Lower Non-Critical    : 1920.000
Upper Non-Critical    : 0.000
Upper Critical        : na
Upper Non-Recoverable : na
Assertion Events      :
Assertions Enabled    :
```

The `--read_power_separators ':' --read_power_previous_token Reading` parameters help parse the output stream, as noted in section 2.5.10.2.6 above.

2.5.12 ESRV Custom Device Libraries

ESRV defines a standard interface to support devices not provided with the Intel® EC SDK. Vendors wishing to add support for their power readers must provide the appropriate libraries for each operating system they want to support:

- a dynamic link library for Windows
- shared object files for Linux, Solaris 10, MacOS X, and MeeGo

Vendors are encouraged to provide support for the entire range of OS types, as well as both 32- and 64-bit versions of ESRV.

The library must implement the six functions listed below.

```
//-----------------------------------------------------------------------------
// functions prototypes
//-----------------------------------------------------------------------------
ESRV_API int init_device_extra_data(PESRV);
ESRV_API int delete_device_extra_data(PESRV);
ESRV_API int open_device(PESRV, void *);
ESRV_API int close_device(PESRV, void *);
ESRV_API int parse_device_option_string(PESRV, void *);
ESRV_API int read_device_power(PESRV, void *, int);
```

2.5.12.1. Hardware Integration of Power

If the device supports hardware integration of power to compute energy, then the following function should be implemented in the library. This function must provide the power reading function in addition to the energy reading function.

```
ESRV_API int read_device_energy(PESRV, void *, int, int);
```

2.5.12.2. Measurement of All Items

If the device measures all the items simultaneously, then the following function should be implemented in the library.

```
ESRV_API int read_device_all_measurements(PESRV, void *, int);
```

2.5.12.3. Measurement of Optional Items

If the device measures any of the optional ESRV items, then the appropriate following functions should be implemented in the library.

```
// Optional functions' prototypes
ESRV_API int read_device_current(PESRV, void *, int);
ESRV_API int read_device_voltage(PESRV, void *, int);
ESRV_API int read_device_power_factor(PESRV, void *, int);
ESRV_API int read_device_voltage_frequency(PESRV, void *, int);
ESRV_API int read_device_current_frequency(PESRV, void *, int);
```

2.5.12.4. Code Templates

`\iecsdk\utils\device_driver_kit\src\energy_meter_driver` is the location in the SDK to find template device code that can be used to build the device library files `esrv_template_device_dynamic_library.c` and `esrv_template_device_dynamic_library.h`. The code is heavily commented and each section of code that may require specific code for a user's device is marked with `// TODO:`.

2.5.12.5. Data Structures

`\iecsdk\src\energy_server\pub_esrv.h` defines a data structure to add the extra data required to manage the device. This data structure is created by ESRV. In the function call flow, ESRV provides two opportunities to initialize the extra data needed to support the device by calling `init_device_extra_data()` function. In between, ESRV calls the `parse_device_option_string()` function, which is an opportunity to update or finalize extra data initialization for the specific device.

For example, the Yokogawa WT210 reader defaults to using the LF (Line Feed) character as its end-of-record marker, but the user can select a different marker (CR or CRLF). If a device option string is provided by the user, then the parser is called and the marker is updated.

NOTE

Dynamically allocated data can be initialized in calls to `init_device_extra_data()`. However, the device library must de-allocate any such resources when ESRV calls the library's `delete_device_extra_data()` function prior to ESRV terminating.

2.5.12.6. Supporting Multiple Device Channels

ESRV defaults to one measurement channel; if the instrument has more than one measurement channel, set the `virtual_devices` variable to the number of measurement channels available. ESRV supports a maximum of ten channels per measurement device.

Similarly, if the measurement device provides hardware integration of power consumption (preferred), then the `f_hw_energy_integration_provided` flag should be changed from its default of zero. In addition, a `read_device_energy()` function must be provided if the flag is changed.

If the device supports simultaneous measurement of all the items and if a `read_device_all_measurements()` function is implemented in the library, then the library should set the `f_optimized_data_read`. Even if the `read_device_all_measurements()` function is found and loaded, it will be called by the ESRV measurement kernel only if the `f_optimized_data_read` flag is set.

Finally, ESRV can also support proprietary interfaces not supported by the base serial ESRV code if `device_interface` is set to `ESRV_DEVICE_INTERFACE_PROPRIETARY` and the appropriate functions to manage that interface are provided; see the documentation in the sample libraries to see how and where this can be done.

The code section below shows how to set the interface flags for a device using a proprietary interface, having three channels, offering harware integration of power overtime, and capable of returning all its measurement in a single read access.

```
//-----------------------------------------------------------------------------
// set default virtual device count (channels)
//-----------------------------------------------------------------------------
p->device_data.virtual_devices = 3;

//-----------------------------------------------------------------------------
// signal hardware energy integration capability
//-----------------------------------------------------------------------------
p->f_hw_energy_integration_provided = 1;

//-----------------------------------------------------------------------------
// signal optimized data measurement capability
//-----------------------------------------------------------------------------
p->f_optimized_data_read = 1;

//-----------------------------------------------------------------------------
// overwrite the default serial interface for library supported devices
//-----------------------------------------------------------------------------
p->device_interface = ESRV_DEVICE_INTERFACE_PROPRIETARY;
```

2.5.12.7. Startup and Process

1. Once the device is opened, the ESRV measurement kernel starts to run.
2. At the requested update interval (one second by default), either the `read_device_power()` or the `read_device_energy()` function is called to collect the data.
3. The measurement kernel then exports the data in the PL counters.
4. After the mandatory data, the kernel reads any supported optional item and exports the data in the PL counters.

The `read_device_energy()` function usually requires support for a set of services (defined in the `\iecsdk\src\energy_server\pub_esrv.h` header file). This is due to the fact that hardware integration may require setup, initialization, resetting, etc. The list below provides a summary of the services that may need to be implemented.

```
READ_DEVICE_ENERGY_INIT: requested once, at server start
READ_DEVICE_ENERGY_START: requested once, at server start right after
READ_DEVICE_ENERGY_READ: requested at will, possibly indefinitely
READ_DEVICE_ENERGY_STOP: requested once at server stop
READ_DEVICE_ENERGY_RESET: requested at will, possibly indefinitely
```

NOTE

This list of services may be expanded in future versions of ESRV.

NOTE

If the meter runs for a very long time, the amount of energy measured could conceivably overrun the meter's internal counters. The library should gracefully handle such occurrences, as indicated in the sample code in section 2.6.2.1.

2.5.12.8. Closing the Device

ESRV calls the `close_device()` and `delete_device_extra_data()` functions in this order when ESRV is interrupted by the user. The `close_device()` function can execute shutdown code if required by the device, including any code required to manage a proprietary interface. The `delete_device_extra_data()` function is where ESRV calls the library to free up dynamically allocated resources prior to program termination.

2.5.12.9. Custom Library Summary

The listing below summarizes the sequence of functions needed in a library to support a custom device in ESRV. Note that calls to individual item measurement functions (except energy) can be coalesced into a single call to `read_device_all_measurements()`.

```
//-----------------------------------------------------------------------------
// function call sequence by the server
//-----------------------------------------------------------------------------
// 1 - .....ESRV_API int init_device_extra_data(PESRV); // first call
// 2 - .....ESRV_API int parse_device_option_string(PESRV, void *);
// 3 - .....ESRV_API int init_device_extra_data(PESRV); // second call
// 4 - .....ESRV_API int open_device(PESRV, void *);
// 5 - N.......ESRV_API int read_device_power(PESRV, void *, int);
//      .......ESRV_API int read_device_energy(PESRV, void *, int); if available
//      .......ESRV_API int read_device_current(PESRV, void *, int);
//      .......ESRV_API int read_device_voltage(PESRV, void *, int);
//      .......ESRV_API int read_device_power_factor(PESRV, void *, int);
//      .......ESRV_API int read_device_current_frequency(PESRV, void *, int);
//      .......ESRV_API int read_device_voltage_frequency(PESRV, void *, int);
// N + 1 - .ESRV_API int close_device(PESRV, void *);
// N + 2 - .ESRV_API int delete_device_extra_data(PESRV);
//-----------------------------------------------------------------------------
```

NOTE

ESRV has been optimized to reduce performance impacts on the system running ESRV. Thus, the default ESRV update interval and the minimum supported interval between samples is one second. Developers should be conscious of the overhead their library incurs; excessive processing within the library could negatively affect the adoption of such libraries by ESRV users.

2.5.13 ESRV Custom DAQ Libraries

Similarly to the power meters, ESRV defines a standard interface to support additional DAQ devices not included in the Intel EC SDK. Vendors wishing to add support for their DAQs must provide the appropriate libraries:

- dynamic link library for Windows
- shared object files for Linux, Solaris 10, MacOS X, and MeeGo.

Vendors are encouraged to provide support for the entire range of OS types, as well as both 32- and 64-bit versions of ESRV.

The library must implement these six functions:

```
//-----------------------------------------------------------------------------
// functions prototype
//-----------------------------------------------------------------------------
ESRV_API int init_device_extra_data(PESRV);
ESRV_API int delete_device_extra_data(PESRV);
ESRV_API int open_device(PESRV, void *);
ESRV_API int close_device(PESRV, void *);
ESRV_API int parse_device_option_string(PESRV, void *);
ESRV_API int read_device_all_channels(PESRV, void *);
ESRV_API int read_device_channel(PESRV, void *, int);
```

With the exception of the last two functions, all other functions in the ESRV DAQ interface are similar to the ones present in the power meter ESRV interface. All these functions, with the exception of `read_device_all_channels`, are ***mandatory***. However, it is highly recommended to implement an optimized channel reading function, such as `read_device_all_channels`, to ensure good performance. Implementing a `read_device_all_channels` function allows ESRV to perform a single function call per reading cycle, rather than numerous calls to `read_device_channel` for each channel in the same reading cycle.

When an optimized function is provided, then the `p->f_daq_optimized_data_read` variable must be set (to 1) during the first call to the `init_device_extra_data` function (described below).

Note that the ESRV DAQ interface doesn't require a `read_device_energy` function, as with the meter devices.

The rest of this section presents the specificities of the ESRV DAQ interface.

2.5.13.1. Optimized Chanel Reading Function

If the device supports an optimized channel reading – to measure all channels simultaneously – then the following function should be implemented in the library. This function must provide the values of all channels. Note that even if the DAQ needs to access one-by-one to its channels, the library writer should bundle all those accessed into an optimized channel-reading function and expose it to ESRV. When defined, ESRV calls only the optimized function, and the `read_device_channel` function can be a stub function.

```
ESRV_API int read_device_all_channels(PESRV, void *);
```

2.5.13.2. Code Templates

`\iecsdk\utils\device_driver_kit\src\energy_meter_driver` is the location in the SDK to find template device code that can be used to build the device library files:

- `esrv_template_daq_device_dynamic_library.c`
- `esrv_template_daq_device_dynamic_library.h`.

The code is heavily commented and each section of code that may require specific code for a user's device is marked with `// TODO:`.

The code snippet below is an extract of a DAQ-simulated ESRV support module, in particular of the `read_device_all_channels` function.

Two important points have to be noted:

1. The number of active channels requested by the user (via the `--channels <n>` option) is available thru the `p->daq_active_channels_count` variable.
2. There is a double indirection to access the memory locations where the channel readings must be stored. A first array provides the list of the active channels (`p->daq_active_channels[]` – int C data type), and a second array stores the channel readings (`p->daq_channel_readings[]` – long double C data type).

ESRV compresses the active channels, so the reading kernel does not spend time running through the DAQ channels list during each cycle reading.

```
//-------------------------------------------------------------------------------
// Note the double indirection of the readings array!
// This is due to the fact that ESRV compresses the active channels table
// as it is expected to not use all channels often and the cost of running
// through the entire table by the ESRV kernel is too expresive in comparision.
// >>>  daq_channel_readings[daq_active_channels[i]]
//-------------------------------------------------------------------------------
for(i = 0; i < p->daq_active_channels_count; i++) {
   p->daq_channel_readings[p->daq_active_channels[i]] =
      5.0 +
      (
         (double)(
            rand() %
            10
         ) /
      10.0
   )
   ;
};
```

2.5.14 Use ESRV to Communicate With a Device

When developing an ESRV support module for a device (power meter or DAQ), it is possible to either communicate with the device directly from the module, or using ESRV. The first option is typically used when the device uses a proprietary interface. The second approach is a facility offered by ESRV, which has built-in support for communication via serial and Ethernet interfaces. This section focuses on the second option and how a module developer can use this facility.

NOTE

For an Ethernet interface, ESRV exposes to the module writer only raw TCP/IP communications (byte transfer). It doesn't expose to the module writer, in this release, any high-level protocols over TCP, such as Telnet, ModBus, VXi11, etc.

Prior to the first call to `init_device_extra_data`, the ESRV driver assumes the device for which the module is being loaded communicates through the serial interface. It is during the first call to `init_device_extra_data` that the module can specify to ESRV to use one of the following interfaces:

- `ESRV_DEVICE_INTERFACE_PROPRIETARY`
- `ESRV_DEVICE_INTERFACE_SERIAL`
- `ESRV_DEVICE_INTERFACE_ETHERNET`

Use `ESRV_DEVICE_INTERFACE_PROPRIETARY` to specify the module will autonomously drive the communication with its device and not require assistance from ESRV.

Use `ESRV_DEVICE_INTERFACE_SERIAL` to indicate to ESRV that the serial interface should be used with this device.

And lastly, use `ESRV_DEVICE_INTERFACE_ETHERNET` to request Ethernet communications with the device (using TCP/IP).

Future releases of ESRV may provide additional support, such as IPMI or GPIB.

After the first call to `init_device_extra_data`, ESRV uses the `device_interface` variable to change, if required, a set of function pointers based on the module request. Once these function pointers are set, it is possible to call them from the support module as shown below.

```
ESRV_API int open_device(PESRV p, void *px) {
...
   ESRV_STATUS = ESRV_FAILURE;
   // these messages are device specific, this is just an example
   char *message = "*RST;*OPC?"
   char *ok = "1"
...
   // setup the message to send to the device
   // note the append of the EOR to the message
   sprintf(
      p->interface_data.output_buffer,
      "%s%s",
      message,
      p->device_data.eor
```

```
    );
    p->interface_data.bytes_to_read = strlen(
       p->interface_data.output_buffer
    );
...
    // send the message to the device
    ret = p->p_write_interface(p);
    if(ret != ESRV_SUCCESS) {
       ; // error
    }
...
    // read and check device's answer
    ret = p->p_read_interface(p);
    if(ret != ESRV_SUCCESS) {
       ; // error
    }
    if(strncmp(
       p->interface_data.input_buffer,
       ok,
       1
    ) != 0) {
       ; // not the expected answer from the device
    }
```

The code snippet above can run over either serial or Ethernet interfaces. ESRV manages the communication specifics. If the Ethernet (TCP) interface is used, the information provided to ESRV via the `--interface_options` string is used to establish the TCP/IP communication (`ip=` and `port=`). Note that the `user=` and `password=` options, if provided via the `--interface_options` string, are made available by ESRV to the module thru the `p->interface_data.user` and `p->interface_data.password` char pointers. However, the login process must be handled by the module code, likely in the `open_device` function.

The functions listed below are available to the module writer for both serial and Ethernet (TCP/IP) communications. Each function requires a pointer to the ERSV data as their single argument (`PESRV p`, which ESRV passes to each module function).

- `p_open_interface`
- `p_close_interface`
- `p_read_interface`
- `p_write_interface`

NOTE

To ensure success in communicating through the interface, before calling `p_write_interface`, *be sure you correctly set up the fields* `p->interface_data.bytes_to_read` *and* `p->device_data.leor` *(the EOR bytes count) using* `init_device_extra_data` *in the device library. If the set value is lower than the actual number of bytes stored in the buffer, then only the required bytes will be sent, and likely not with the EOR bytes. This will make the device fail to correctly analyze the sent message. If the number of bytes to write is larger than the number of actual bytes stored in the buffer – including the EOR bytes – then the transmission will stop when the EOR bytes are sent.*

NOTE

The `p->interface_data.input_buffer` *contains the bytes sent by the device – excluding the EOR bytes – after returning from the* `p_read_interface` *function.* `p->interface_data.bytes_read` *is also set accordingly.*

NOTE

The following variables (`int`*) are available to the module developer for debugging ESRV/device communications. These variables can be accessed via the* `p->interface_data.x` *pointer (replace* `x` *by one of the names listed below).*

- `bytes_to_read`
- `bytes_to_write`
- `bytes_read`
- `bytes_written`

2.6 Using ESRV Counters

This section describes all the available ESRV counters.

2.6.1 ESRV Exported Counters

ESRV uses the Intel EC API to export the counters listed below:

```
Energy (Joule)
Energy (Joule).decimals
Energy (kWh)
Energy (kWh).decimals
Energy Overflows (no unit)
Update Frequency (second)
Power (Watt)
Power (Watt).decimals
Power (Watt)--Max
Power (Watt)--Max.decimals
Power (Watt)--Min
Power (Watt)--Min.decimals
Current (Ampere)
Current (Ampere).sign
Current (Ampere).decimals
Current (Ampere)--Max
Current (Ampere)--Max.sign
Current (Ampere)--Max.decimals
Current (Ampere)--Min
Current (Ampere)--Min.sign
Current (Ampere)--Min.decimals
Current (Ampere seconds)
Current (Ampere seconds).sign
Current (Ampere seconds).decimals
Voltage (Volt)
Voltage (Volt).sign
Voltage (Volt).decimals
Voltage (Volt)--Max
```

```
29   Voltage (Volt)--Max.sign
30   Voltage (Volt)--Max.decimals
31   Voltage (Volt)--Min
32   Voltage (Volt)--Min.sign
33   Voltage (Volt)--Min.decimals
34   Voltage (Volt seconds)
35   Voltage (Volt seconds).sign
36   Voltage (Volt seconds).decimals
37   Power Factor (no unit)
38   Power Factor (no unit).decimals
39   Power Factor (no unit)--Max
40   Power Factor (no unit)--Max.decimals
41   Power Factor (no unit)--Min
42   Power Factor (no unit)--Min.decimals
43   Power Factor (no unit seconds)
44   Power Factor (no unit seconds).decimals
45   Current Frequency (Hertz)
46   Current Frequency (Hertz).decimals
47   Current Frequency (Hertz)--Max
48   Current Frequency (Hertz)--Max.decimals
49   Current Frequency (Hertz)--Min
50   Current Frequency (Hertz)--Min.decimals
51   Current Frequency (Hertz seconds)
52   Current Frequency (Hertz seconds).decimals
53   Voltage Frequency (Hertz)
54   Voltage Frequency (Hertz).decimals
55   Voltage Frequency (Hertz)--Max
56   Voltage Frequency (Hertz)--Max.decimals
57   Voltage Frequency (Hertz)--Min
58   Voltage Frequency (Hertz)--Min.decimals
59   Voltage Frequency (Hertz seconds)
60   Voltage Frequency (Hertz seconds).decimals
61       Channel(s)
62       Status
63       Version
```

NOTE

Counters listed in bold above (lines 1-12 and 61-63) are always present and fed with data in any ESRV dataset. Other counters may not be supported by all devices. If an optional counter is not supported, ESRV sets that counter to zero. Note that none of these optional counters is required to perform an energy efficiency analysis.

2.6.1.1. Integral Counters

Energy is an integration of power (a rate) over time. By reading the energy counter at two arbitrary times and subtracting the difference, developers can get the average power consumption for that interval.

For example, assume an application needs to determine average system power consumption over a two-hour benchmark run. One method would be to record power every second to determine the average power consumption. A simpler method would be to read the `Energy (Joule)` counter at the start of the benchmark, subtract that from the `Energy (Joule)` counter reading at the end of the benchmark, and divide the difference by the number of seconds in between.

This works fine for determining the average power draw, but what about the average voltage, average current draw, or average power factor? ESRV provides several other integral counters, such as `Current (Ampere seconds)`, `Voltage (Volt seconds)`, and `Power Factor (no unit seconds)`. These integral counters can be used to determine averages over any given period of time while ESRV is continuously sampling data.

NOTE

Even overlapping averages can be computed from the ESRV integral counters by reading the integral counters at the start and end of period, subtracting to get the difference, and then dividing by the number of seconds between the two readings.

2.6.1.2. Energy

The energy counters are equal to the amount of energy (in J and kWh) the monitored system consumes since the last counter reset or ESRV startup. All these counters are mandatory, which means that any ESRV data set will provide them to client applications using the Intel EC API.

The **`Energy (Joule)`** counter is provided as a fixed decimal floating-point value. By convention, floating point values are represented by two counter values: the actual counter plus a static second counter with a `.decimals` suffix representing the number of digits to the right of the decimal. Since the `Energy (Joule).decimals` counter is set to two, this indicates that the `Energy (Joule)` counter value is 100 times (10^2 times) the real value. For example, an `Energy (Joule)` value of 6351 would actually represent 63.51 joules.

NOTE

One joule equals one watt-second (1 J = 1 W-s).

The **`Energy in kWh`** counter provides a fixed decimal floating-point value. Since the `Energy (kWh).decimals` counter is set to two, this indicates that the `Energy (kWh)` counter value is 100 times (10^2 times) the real value. For example, an `Energy (kWh)` value of 1536 would actually represent 15.36 kWh.

The **`Energy Overflows`** counter provides the number of times the energy counter (in joules) has overflowed. See section 2.2.4 for more information on the range of ESRV counters.

2.6.1.3. Power

The **`Power (Watt)`** counter provides the power measured during last sampling interval. Since `Power (Watt).decimals` is set to two, this indicates that the `Power` counter value represents 100 times (10^2 times) the actual value.

The **`Power (Watt)--Max`** counter provides the maximum power measured since ESRV startup. Since `Power (Watt)--Max.decimals` is set to two, this indicates that the `Power` counter value represents 100 times (10^2 times) the actual value.

The **`Power (Watt)--Min`** counter provides the minimum power measured since ESRV startup. Since `Power (Watt)--Min.decimals` is set to two, this indicates that the `Power` counter value represents 100 times (10^2 times) the actual value.

The **`Power Factor (no unit)`** counter provides the power factor measured during last sampling interval. Since `Power Factor (no unit).decimals` is set to four, then the `Power Factor` counter value represents 10,000 times (10^4 times) the actual value.

NOTE

Power factor is the ratio of the actual power dissipated in an electrical system to the input power of volts multiplied by amps. Computer power supplies and other non-linear loads may distort the wave shape of the power drawn, rather than consuming power in nice sine waves like most resistive loads. All other things being equal, a power supply with a higher power factor is more efficient than one with a lower power factor.

The **`Power Factor (no unit)--Max`** counter provides the maximum power factor measured since ESRV startup. Since `Power Factor (no unit)--Max.decimals` is set to four, then the `Power Factor` counter value represents 10,000 times (10^4 times) the actual value.

The **`Power Factor (no unit)--Min`** counter provides the minimum power factor measured since ESRV startup. Since `Power Factor (no unit)--Min.decimals` is set to four, then the `Power Factor` counter value represents 10,000 times (10^4 times) the actual value.

The **`Power Factor (no unit seconds)`** counter provides the power factor integral over time since ESRV startup. Since `Power Factor (no unit seconds).decimals` is set to four, then the `Power Factor (no unit seconds)` counter value represents 10,000 times (10^4 times) the actual value.

2.6.1.4. Current

The **`Current (Ampere)`** counter provides the current measured during last sampling interval. Since the `Current (Ampere).decimals` counter is set to four, this indicates that the `Current (Ampere)` counter value represents 10,000 times (10^4 times) the actual value.

The **`Current (Ampere).sign`** counter indicates the sign of the current. By convention, a non-zero value indicates a negative current. It is positive if the counter is null.

The **`Current (Ampere)--Max`** counter provides the maximum current measured since ESRV startup. Since `Current (Ampere)--Max.decimals` is set to four, this indicates that the `Current (Ampere)--Max` counter value represents 10,000 times (10^4 times) the actual value.

The **`Current (Ampere)--Max.sign`** counter indicates the sign of the current. By convention, a non-zero value indicates a negative current. It is positive if the counter is null.

The **`Current (Ampere)--Min`** counter provides the minimum current measured since ESRV startup. Since `Current (Ampere)--Min.decimals` is set to four, this indicates that the `Current (Ampere)--Max` counter value represents 10,000 times (10^4 times) the actual value.

The **Current (Ampere)--Min.sign** counter indicates the sign of the current. By convention, a non-zero value indicates a negative current. It is positive if the counter is null.

The **Current (Ampere seconds)** counter provides the current integral over time since ESRV startup. Since `Current (Ampere seconds).decimals` is set to four, this indicates that the `Current (Ampere seconds)` counter value represents 10,000 times (10^4 times) the actual value.

The **Current (Ampere seconds).sign** counter indicates the sign of the current integral. By convention, a non-zero value indicates a negative current integral. It is positive if the counter is null.

The **Current Frequency (Hertz)** counter provides the current frequency measured during last sampling interval. Since `Current Frequency (Hertz).decimals` is set to four, then the `Current Frequency (Hertz)` counter value represents 10,000 times (10^4 times) the actual value.

NOTE

Most meters measure voltage frequency, but only a subset of meters measure current frequency.

The **Current Frequency (Hertz)--Max** counter provides the current maximum frequency measured since ESRV startup. Since `Current Frequency (Hertz)--Max.decimals` is set to four, then the `Current Frequency (Hertz)--Max` counter value represents 10,000 times (10^4 times) the actual value.

The **Current Frequency (Hertz)--Min** counter provides the current maximum frequency measured since ESRV startup. Since `Current Frequency (Hertz)--Min.decimals` is set to four, then the `Current Frequency (Hertz)--Min` counter value represents 10,000 times (10^4 times) the actual value.

The **Current Frequency (Hertz seconds)** counter is the current frequency integral over time since ESRV startup. Since `Current Frequency (Hertz seconds).decimals` is set to four, then the `Current Frequency (Hertz seconds)` counter value represents 10,000 times (10^4 times) the actual value.

2.6.1.5. Voltage

The **Voltage (Volt)** counter provides the voltage measured during last sampling interval. Since the named `Voltage (Volt).decimals` counter is set to four, then the `Voltage (Volt)` counter value represents 10,000 times (10^4 times) the actual value.

The **Voltage (Volt).sign** counter indicates the sign of the voltage. By convention, a non-zero value indicates a negative voltage. It is positive if the counter is null.

The **Voltage (Volt)--Max** counter provides the maximum voltage measured since ESRV startup. Since the `Voltage (Volt)--Max.decimals` counter is set to four, then the `Voltage (Volt)--Max` counter value represents 10,000 times (10^4 times) the actual value.

The **Voltage (Volt)--Max.sign** counter indicates the sign of the voltage. By convention, a non-zero value indicates a negative voltage. It is positive if the counter is null.

The **Voltage (Volt)--Min** counter provides the minimum voltage measured since ESRV startup. Since the `Voltage (Volt)--Min.decimals` counter value is set to four, then the `Voltage (Volt)--Min` counter represents 10,000 times (10^4 times) the actual value.

The **Voltage (Volt)--Min.sign** counter indicates the sign of the voltage. By convention, a non-zero value indicates a negative voltage. It is positive if the counter is null.

The **Voltage (Volt seconds)** counter provides the voltage integral over time since ESRV startup. Since the `Voltage (Volt seconds).decimals` counter is set to four, then the `Voltage (volt seconds)` counter value represents 10,000 times (10^4 times) the actual value.

The **Voltage (Volt seconds).sign** counter indicates the sign of the voltage integral. By convention, a non-zero value indicates a negative voltage. It is positive if the counter is null.

The **Voltage Frequency (Hertz)** counter provides the voltage frequency measured during last sampling interval. Since `Voltage Frequency (Hertz).decimals` is set to four, then the `Voltage Frequency` counter value represents 10,000 times (10^4 times) the actual value.

The **Voltage Frequency (Hertz)--Max** counter provides the voltage maximum frequency measured since ESRV startup. Since `Voltage Frequency (Hertz)--Max.decimals` is set to four, then the `Voltage Frequency (Hertz)--Max` counter value represents 10,000 times (10^4 times) the actual value.

The **Voltage Frequency (Hertz)--Min** counter provides the voltage maximum frequency measured since ESRV startup. Since `Voltage Frequency (Hertz)--Min.decimals` is set to four, then the `Voltage Frequency (Hertz)--Min` counter value represents 10,000 times (10^4 times) the actual value.

The **Voltage Frequency (Hertz seconds)** counter provides the voltage frequency integral over time since ESRV startup. Since the integral counter named `Voltage Frequency (Hertz seconds).decimals` is set to four, then the `Voltage Frequency (Hertz seconds)` counter value represents 10,000 times (10^4 times) the actual value.

2.6.1.6. Miscellaneous

The **Update Frequency** counter provides the number of seconds between samples (one second by default).

The **channel(s)** counter indicates the number of measurement channel(s) provided by the measurement device used by ESRV.

The **Status** counter indicates if ESRV is running or not. ESRV sets this counter to a non-zero value while running and then resets it to zero when terminating.

The **Version** counter encodes the ESRV version as its value (ESRV versions are in YYYYMMDD form).

2.6.1.7. Multiple Channels

If the measuring devices supports multiple channels, then the set of counters (mandatory and optional) is duplicated and appropriately prefixed as shown in the subset of counters below:

```
[CHANNEL1] - Energy (Joule)
[CHANNEL1] - Energy (Joule).decimals
[CHANNEL1] - Energy (kWh)
[CHANNEL1] - Energy (kWh).decimals
[CHANNEL1] - Power (Watt)
[CHANNEL1] - Power (Watt).decimals
...
[CHANNEL2] - Energy (Joule)
[CHANNEL2] - Energy (Joule).decimals
[CHANNEL2] - Energy (kWh)
[CHANNEL2] - Energy (kWh).decimals
[CHANNEL2] - Power (Watt)
[CHANNEL2] - Power (Watt).decimals
...
[CHANNEL3] - Energy (Joule)
[CHANNEL3] - Energy (Joule).decimals
[CHANNEL3] - Energy (kWh)
[CHANNEL3] - Energy (kWh).decimals
[CHANNEL3] - Power (Watt)
[CHANNEL3] - Power (Watt).decimals
```

2.6.2 Reading ESRV Counters

This section explains how applications can use ESRV counters to report energy efficiency. More details are provided in the *Intel Energy Checker SDK User Guide*. The goal of this section is to provide developers of device support libraries an overview of how the application uses the data they provide.

To facilitate the use of ESRV counters using the Intel EC API, Intel Intel EC SDK provides a header file (`pub_esrv_counters.h` file in the `\iecsdk\src\energy_server` folder). Include this file in applications that interface to ESRV. The extract from `pub_esrv_counters.h` shown below contains counter index definitions to be used with `pl_read()` API function calls.

```
//-----------------------------------------------------------------------------
// counters definitions.
//-----------------------------------------------------------------------------
typedef enum _esrv_counters_base_indexes {

   //--------------------------------------------------------------------------
   // mandatory counters
   //--------------------------------------------------------------------------
   ESRV_COUNTER_ENERGY_JOULES_INDEX = 0,
   ESRV_COUNTER_ENERGY_JOULES_DECIMALS_INDEX,
   ESRV_COUNTER_ENERGY_KWH_INDEX,
   ESRV_COUNTER_ENERGY_KWH_DECIMALS_INDEX,
   ESRV_COUNTER_ENERGY_OVERFLOWS_INDEX,
   ESRV_COUNTER_UPDATE_FREQUENCY_INDEX,
   ESRV_COUNTER_POWER_INDEX,
   ESRV_COUNTER_POWER_DECIMALS_INDEX,
   ESRV_COUNTER_MAX_POWER_INDEX,
   ESRV_COUNTER_MAX_POWER_DECIMALS_INDEX,
   ESRV_COUNTER_MIN_POWER_INDEX,
   ESRV_COUNTER_MIN_POWER_DECIMALS_INDEX,

   //--------------------------------------------------------------------------
```

```
    // optional counters
    // Note: counters below are optional and may not be updated or set by
    // the device driver.  It is recommended that un-implemented counters
    // are set to zero.
    //------------------------------------------------------------------------
    ESRV_COUNTER_CURRENT_INDEX,
    ESRV_COUNTER_CURRENT_SIGN_INDEX,
    ESRV_COUNTER_CURRENT_DECIMALS_INDEX,
    ESRV_COUNTER_MAX_CURRENT_INDEX,
    ESRV_COUNTER_MAX_CURRENT_SIGN_INDEX,
    ESRV_COUNTER_MAX_CURRENT_DECIMALS_INDEX,
    ESRV_COUNTER_MIN_CURRENT_INDEX,
    ESRV_COUNTER_MIN_CURRENT_SIGN_INDEX,
    ESRV_COUNTER_MIN_CURRENT_DECIMALS_INDEX,
    ESRV_COUNTER_CURRENT_SECONDS_INDEX,
    ESRV_COUNTER_CURRENT_SECONDS_SIGN_INDEX,
    ESRV_COUNTER_CURRENT_SECONDS_DECIMALS_INDEX,
    ESRV_COUNTER_VOLTAGE_INDEX,
    ESRV_COUNTER_VOLTAGE_SIGN_INDEX,
    ESRV_COUNTER_VOLTAGE_DECIMALS_INDEX,
    ESRV_COUNTER_MAX_VOLTAGE_INDEX,
    ESRV_COUNTER_MAX_VOLTAGE_SIGN_INDEX,
    ESRV_COUNTER_MAX_VOLTAGE_DECIMALS_INDEX,
    ESRV_COUNTER_MIN_VOLTAGE_INDEX,
    ESRV_COUNTER_MIN_VOLTAGE_SIGN_INDEX,
    ESRV_COUNTER_MIN_VOLTAGE_DECIMALS_INDEX,
    ESRV_COUNTER_VOLTAGE_SECONDS_INDEX,
    ESRV_COUNTER_VOLTAGE_SECONDS_SIGN_INDEX,
    ESRV_COUNTER_VOLTAGE_SECONDS_DECIMALS_INDEX,
    ESRV_COUNTER_POWER_FACTOR_INDEX,
    ESRV_COUNTER_POWER_FACTOR_DECIMALS_INDEX,
    ESRV_COUNTER_MAX_POWER_FACTOR_INDEX,
    ESRV_COUNTER_MAX_POWER_FACTOR_DECIMALS_INDEX,
    ESRV_COUNTER_MIN_POWER_FACTOR_INDEX,
    ESRV_COUNTER_MIN_POWER_FACTOR_DECIMALS_INDEX,
    ESRV_COUNTER_POWER_FACTOR_SECONDS_INDEX,
    ESRV_COUNTER_POWER_FACTOR_SECONDS_DECIMALS_INDEX,
    ESRV_COUNTER_CURRENT_FREQUENCY_INDEX,
    ESRV_COUNTER_CURRENT_FREQUENCY_DECIMALS_INDEX,
    ESRV_COUNTER_MAX_CURRENT_FREQUENCY_INDEX,
    ESRV_COUNTER_MAX_CURRENT_FREQUENCY_DECIMALS_INDEX,
    ESRV_COUNTER_MIN_CURRENT_FREQUENCY_INDEX,
    ESRV_COUNTER_MIN_CURRENT_FREQUENCY_DECIMALS_INDEX,
    ESRV_COUNTER_CURRENT_FREQUENCY_SECONDS_INDEX,
    ESRV_COUNTER_CURRENT_FREQUENCY_SECONDS_DECIMALS_INDEX,
    ESRV_COUNTER_VOLTAGE_FREQUENCY_INDEX,
    ESRV_COUNTER_VOLTAGE_FREQUENCY_DECIMALS_INDEX,
    ESRV_COUNTER_MAX_VOLTAGE_FREQUENCY_INDEX,
    ESRV_COUNTER_MAX_VOLTAGE_FREQUENCY_DECIMALS_INDEX,
    ESRV_COUNTER_MIN_VOLTAGE_FREQUENCY_INDEX,
    ESRV_COUNTER_MIN_VOLTAGE_FREQUENCY_DECIMALS_INDEX,
    ESRV_COUNTER_VOLTAGE_FREQUENCY_SECONDS_INDEX,
    ESRV_COUNTER_VOLTAGE_FREQUENCY_SECONDS_DECIMALS_INDEX,

    //------------------------------------------------------------------------
    // esrv specific counters
    //------------------------------------------------------------------------
    ESRV_COUNTER_CHANNELS_INDEX,
```

```
   ESRV_COUNTER_STATUS_INDEX,
   ESRV_COUNTER_VERSION_INDEX

} ESRV_COUNTERS_BASE_INDEXES;
```

2.6.2.1. ESRV Sample Code

The following code snippets show a typical use of ESRV counters in a C program to report energy efficiency from an application. Note that this code can be implemented as a thread or as a process if the work data can be exchanged via an IPC mechanism.

In this sample, the application connects automatically to the latest ESRV instance. Alternatively, the developer could allow the user to identify the ESRV instance to use via a dialog box, command line argument, or similar mechanism.

The following sample code computes and reports energy efficiency based on data collected during one sample period. An alternate approach is to use cumulative measurements of work and energy and then report average energy efficiency.

NOTE

The ESRV client sample code shipped with the SDK (`iecsdk\src\samples\esrv_client`*) provides both implementations discussed above. Please refer to the sample code for more details. This sample code also shows how to use other ESRV counters such as* `Channel(s)`, `Status` *and* `.decimals` *suffix counters.*

```
#include "productivity_link.h"
#include "productivity_link_helper.h"
#include "pub_esrv_counters.h"

#define COUNTERS_COUNT 5
#define EE_COUNTERS_DECIMALS 4
#define EE_SCALE 10000.0 // 10 ^ 4

int main(void) {

...

   uuid_t uuid;
   char application_name[] = "My Application";
   const char *counters_names[COUNTERS_COUNT] = {
      "Work Units Done",
      "Energy Consumed (in Joules)",
      "Joule(s) per Work Unit",
      "Joule(s) per Work Unit.decimals",
      "Work Unit(s) per Joule",
      "Work Unit(s) per Joule.decimals"
   };

   enum {
      WORK_UNITS_DONE_INDEX = 0,
      ENERGY_CONSUMED_INDEX,
      JOULES_PER_WORK_UNIT_INDEX,
      JOULES_PER_WORK_UNIT_DECIMALS_INDEX,
      WORK_UNIT_PER_JOULE_INDEX,
      WORK_UNIT_PER_JOULE_DECIMALS_INDEX
   };
```

```

   unsigned long long energy = 0;
   unsigned long long consumed_energy = 0;
   unsigned long long reference_energy = 0;
   unsigned long long work_units = 0;
   unsigned long long work_units_done = 0;
   unsigned long long reference_work_units = 0;
   double joules_per_work_unit = 0.0;
   double work_unit_per_joule = 0.0;
   unsigned long long value = 0;

   char esrv_pl_config_file [MAX_PATH] = { 0 };
   int ret = PL_FAILURE;
   int pl_esrv = PL_INVALID_DESCRIPTOR;
   int pld = PL_INVALID_DESCRIPTOR;

...

   //---------------------------------------------------------------------------
   // attach to the latest ESRV instance
   //---------------------------------------------------------------------------
   memset(esrv_pl_config_file, 0, sizeof(esrv_pl_config_file));
   ret = plh_get_young_pl_by_application_name("esrv", &esrv_pl_config_file[0]);
   assert(ret != PL_FAILURE);
   pl_esrv = pl_attach(file_energy);
   assert(pl_esrv != PL_INVALID_DESCRIPTOR);

   //---------------------------------------------------------------------------
   // open application's PL
   //---------------------------------------------------------------------------
   pld = pl_open(application_name, COUNTERS_COUNT, counters_names, &uuid);
   assert(pld != PL_INVALID_DESCRIPTOR);

   //---------------------------------------------------------------------------
   // write-out static counters
   //---------------------------------------------------------------------------
   value = EE_COUNTERS_DECIMALS;
   ret = pl_write(pld, &value, JOULES_PER_WORK_UNIT_DECIMALS_INDEX);
   assert(ret != PL_FAILURE);
   ret = pl_write(pld, &value, WORK_UNIT_PER_JOULE_DECIMALS_INDEX);
   assert(ret != PL_FAILURE);

...

   //---------------------------------------------------------------------------
   // read the reference energy
   //---------------------------------------------------------------------------
   ret = pl_read(pl_esrv, &reference_energy, ESRV_COUNTER_ENERGY_JOULES_INDEX);
   assert(ret != PL_FAILURE);

...

   while(1) {

   ...

      //------------------------------------------------------------------------
      // read the instantaneous energy
```

```
      //-------------------------------------------------------------------------
      ret = pl_read(pl_esrv, &energy, ESRV_COUNTER_ENERGY_JOULES_INDEX);
      assert(ret != PL_FAILURE);

      //-------------------------------------------------------------------------
      // compute energy consumed and useful work units done
      //-------------------------------------------------------------------------
      consumed_energy = energy - reference_energy;
      // colect the work units done (assumed monotonously increasing)
      work_units_done = work_units - reference_work_units;

      //-------------------------------------------------------------------------
      // write-out work units done and energy consumed
      //-------------------------------------------------------------------------
      ret = pl_write(pld, &consumed_energy, ENERGY_CONSUMED_INDEX);
      assert(ret != PL_FAILURE);
      ret = pl_write(pld, & work_units_done, WORK_UNITS_DONE_INDEX);
      assert(ret != PL_FAILURE);

      //-------------------------------------------------------------------------
      // compute and write-out joules per work unit EE counter
      //-------------------------------------------------------------------------
      if(work_units_done != 0) {
         joules_per_work_unit =
            (double)consumed_energy / (double)work_units_done;
      } else {
         joules_per_work_unit = 0.0;
      }
      value = (unsigned long long)(joules_per_work_unit * EE_SCALE);
      ret = pl_write(pld, &value, JOULES_PER_WORK_UNIT_INDEX);
      assert(ret != PL_FAILURE);

      //-------------------------------------------------------------------------
      // compute and write-out work unit per joule EE counter
      //-------------------------------------------------------------------------
      if(consumed_energy != 0) {
         work_unit_per_joule =
            (double)work_units_done / (double)consumed_energy;
      } else {
              work_unit_per_joule = 0.0;
      }
      value = (unsigned long long)(work_unit_per_joule * EE_SCALE);
      ret = pl_write(pld, &value, JOULES_PER_WORK_UNIT_INDEX);
      assert(ret != PL_FAILURE);

   ...

      //-------------------------------------------------------------------------
      // save reference work units done and energy for next sample
      //-------------------------------------------------------------------------
      reference_work_units = work_units;
      reference_energy = energy;

   ...

   }

...
```

```

}
```

2.6.3 Using ESRV Programmatically

It may be valuable to programmatically run ESRV from an application. Indeed, if a software developer writes energy-aware code, that code needs to import data from the energy counter of a running ESRV instance, which measures the energy consumption of the system on which the code is running. The sample code below shows how to perform such a task.

The sample assumes that an environment variable (ESRV_GUID) is set when an ESRV instance is already running. Otherwise, it will start an ESRV instance during initialization. During the application's termination, it will stop the ESRV instance and remove its PL from the system's PL_FOLDER.

A complete application (energy) using this mechanism can also be found in the SDK samples.

2.6.3.1. Header File:

```
//-----------------------------------------------------------------------------
// ESRV pid and uuid retrieval
//-----------------------------------------------------------------------------
#define ENERGY_INPUT_LINE_MAX_SIZE 4096
#define ENERGY_ESRV_GUID_LINE 9
#define ENERGY_ESRV_PID_LINE 10
#define ENERGY_GUID_LENGHT_IN_CHARACTERS 36
#define ENERGY_ESRV_PRE_GUID_TOKEN "Using Guid:     "
#define ENERGY_ESRV_GUID_TOKEN_SEPARATORS "\n["
#define ENERGY_ESRV_GUID_TOKEN_TERMINATOR "]"
#define ENERGY_ESRV_PRE_PID_TOKEN "PID:            "
#define ENERGY_ESRV_PID_TOKEN_SEPARATORS "\n["
#define ENERGY_ESRV_PID_TOKEN_TERMINATOR "]"

//-----------------------------------------------------------------------------
// ESRV shell variables
//-----------------------------------------------------------------------------
#define ENERGY_ESRV_DEFAULT_GUID_SHELL_VARIABLE "ESRV_GUID"

//-----------------------------------------------------------------------------
// ESRV instance handling
//-----------------------------------------------------------------------------
#define ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE "_"
#ifdef __PL_WINDOWS__
   #define ENERGY_ESRV_PL_CONFIG_FILE_NAME "\\pl_config.ini"
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
   #define ENERGY_ESRV_PL_CONFIG_FILE_NAME "/pl_config.ini"
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
#define ENERGY_ESRV_PL_CONFIG_FILE_APPLICATION_NAME "esrv_"
#define ENERGY_ESRV_BINARY_NAME "esrv --start "
#define ENERGY_ESRV_SHELL_OPTION " --shell"
#ifdef __PL_WINDOWS__
   #define ENERGY_ESRV_PL_CONFIG_FILE_ROOT "C:\\productivity_link\\"
#endif // __PL_WINDOWS__
```

```
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
   #define ENERGY_ESRV_PL_CONFIG_FILE_ROOT "/opt/productivity_link/"
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
#if defined (_DEBUG) || (__PL_DEBUG__)
   #ifdef __PL_WINDOWS__
      #define ENERGY_ESRV_DEFAULT_OPTIONS "--library esrv_simulated_device.dll"
   #endif // __PL_WINDOWS__
   #if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
      #define ENERGY_ESRV_DEFAULT_OPTIONS "--library esrv_simulated_device.so"
   #endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
#else // _DEBUG || __PL_DEBUG__
   #define ENERGY_ESRV_DEFAULT_OPTIONS "--device y210 --device_options \"items=all\""
#endif // _DEBUG || __PL_DEBUG__

typedef struct _energy_data {

   //-------------------------------------------------------------------------
   // Parser data.
   //-------------------------------------------------------------------------
   int argc;
   char **argv;

   int f_interrupted;
   int f_guid;
   int f_guid_shell_variable;
   int f_channel;
   int f_esrv_options;
   int f_version;
   int f_help;
   int f_command;

   //--------------------------------------------------------------------------
   // Data to manage ESRV instance.
   //--------------------------------------------------------------------------
   FILE *fp_esrv;

   //--------------------------------------------------------------------------
   // Data to manage ESRV PL.
   //--------------------------------------------------------------------------
   char *esrv_guid;
   char *esrv_guid_shell_variable;
   char *esrv_guid_shell_variable_value;
   char *esrv_options;

   //--------------------------------------------------------------------------
   // Data to manage the attached ESRV.
   //--------------------------------------------------------------------------
   unsigned int channel;
   unsigned long long channels;
   unsigned long long status;
   unsigned long long version;

   //-------------------------------------------------------------------------
   // Energy sampling data.
   //-------------------------------------------------------------------------
   int esrv_pld;
   unsigned long long start_energy_data;
   unsigned long long end_energy_data;
```

```
   double consumed_energy_in_joules;
   double consumed_energy_in_kwhs;
   double energy_data_multiplier;
   unsigned long long esrv_status;

   //-------------------------------------------------------------------------
   // Data to manage command.
   //-------------------------------------------------------------------------
   char command[MAX_PATH];
   FILE *fp;

} ENERGY_DATA, *PENERGY_DATA;

//-----------------------------------------------------------------------------
// macros
//-----------------------------------------------------------------------------
#define _ERROR(s) \
   fprintf(stderr, "Error: %s [%d]\n", s, __LINE__); \
   goto error;

#define ERROR_INDEX(s, t, i) \
   memset(buffer, 0, sizeof(buffer)); \
   sprintf(buffer, s, (t), (i)); \
   fprintf(stderr, "Error: %s [%d]\n", buffer, __LINE__); \
   goto error;

#define WARNING(s) \
   fprintf(stdout, "Warning: %s [%d]\n", s, __LINE__);

//-----------------------------------------------------------------------------
// function prototypes
//-----------------------------------------------------------------------------
extern int main(int argc, char *argv[], char *envp[]);
extern int parser(PENERGY_DATA);
#ifdef __PL_WINDOWS__
BOOL signal_handler(DWORD);
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
void signal_handler(int);
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
```

2.6.3.2. Code File:

```
#include <assert.h>
#include <stdio.h>
#include <math.h> // for pow
#ifdef __PL_WINDOWS__
   #include <io.h>
   #include <windows.h>
   #include <tchar.h>
   #include <direct.h>
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
   #include <signal.h> // compile with -std=gnu99 not with -std=c99
   #include <string.h> // for memset
```

```
   #include <unistd.h> // for sleep
   #include <uuid/uuid.h>
   #include <sys/types.h>
   #include <dirent.h>
   #include <ctype.h>
   #include <math.h> // for pow
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
#include "productivity_link.h"
#include "productivity_link_helper.h"
#include "pub_esrv_counters.h"
#include "energy.h"

//-----------------------------------------------------------------------------
// Global so the signal handler can access it.
//-----------------------------------------------------------------------------
ENERGY_DATA energy_data;

//-----------------------------------------------------------------------------
// program entry point.
//-----------------------------------------------------------------------------
int main(int argc, char *argv[], char *envp[]) {

   //--------------------------------------------------------------------------
   // Generic variables.
   //--------------------------------------------------------------------------
   int ret = -1;
   PENERGY_DATA p = NULL;
   PL_STATUS plret = PL_FAILURE;
   unsigned long long value = 0;
   char buffer[ENERGY_INPUT_LINE_MAX_SIZE] = { '\0' };
#ifdef __PL_WINDOWS__
   BOOL bret = FALSE;
#endif // __PL_WINDOWS__

   //--------------------------------------------------------------------------
   // ESRV instance handling variables.
   //--------------------------------------------------------------------------
   char esrv_config_file_name_buffer[MAX_PATH] = { '\0' };
   int f_esrv_guid = 0;
   int f_esrv_pid = 0;
   char *env = NULL;

   //--------------------------------------------------------------------------
   // Variables used to build ESRV start command -- if we have to.
   //--------------------------------------------------------------------------
   char esrv_command_buffer[MAX_PATH] = { '\0' };

   //--------------------------------------------------------------------------
   // Variables used to remove ESRV PL after use -- if we started it.
   //--------------------------------------------------------------------------
   char esrv_pl_path_name[MAX_PATH] = { '\0' };
#ifdef __PL_WINDOWS__
   char current_path_name[MAX_PATH] = { '\0' };
   char esrv_pl_config_file_name[MAX_PATH] = { '\0' };
   struct _finddata_t find_files;
   intptr_t file = 0;
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
```

```
   DIR *directory = NULL;
   struct dirent *file = NULL;
   char file_name[MAX_PATH] = { '\0' };
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__

   //-----------------------------------------------------------------------
   // Variables used to read ESRV's sartup output -- used to get the GUID
   //-----------------------------------------------------------------------
   char *pc = NULL;
   char *token = NULL;
   int input_line_count = 0;
   char esrv_guid[ENERGY_GUID_LENGHT_IN_CHARACTERS + 1] = { '\0' };
#ifdef __PL_WINDOWS__
   DWORD esrv_pid = 0;
   HANDLE esrv_handle = NULL;
   DWORD bytes_read = 0;
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
   pid_t esrv_pid = 0;
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__

   //-----------------------------------------------------------------------
   // Signal handler variables.
   //-----------------------------------------------------------------------
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
   struct sigaction sa;
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__

   //------------------------------------------------------------------------

   //-----------------------------------------------------------------------
   // Retrive the energy structure address.
   //-----------------------------------------------------------------------
   p = &energy_data;
   assert(p != NULL);

   //-----------------------------------------------------------------------
   // Initialize the energy data structure.
   //-----------------------------------------------------------------------
   memset(p, 0, sizeof(ENERGY_DATA));
   p->argc = argc;
   p->argv = argv;
   p->channel = ENERGY_DEFAULT_CHANNEL;
   p->esrv_pld = PL_INVALID_DESCRIPTOR;
   p->esrv_status = ESRV_STATUS_NOT_RUNNING;

   //-----------------------------------------------------------------------
   // Parse user input.
   //-----------------------------------------------------------------------
   plret = parser(&energy_data);
   if(plret != PL_SUCCESS) {
      _ERROR("Unable to make sense of user input.");
   } else {
      if((p->f_help == 1) || (p->f_version == 1)) {
         goto done;
      }
   }

```

```
   //-------------------------------------------------------------------------
   // Check for inconsistent user input.
   //-------------------------------------------------------------------------
   if(
      (
         (p->f_guid == 1) &&
         (p->f_guid_shell_variable)
      )
   ) {
      _ERROR(
         "Incompatible ESRV instance designation.  Please use --guid or --
guid_shell_variable option."
      );
   }
   if(p->f_command == 0) {
      _ERROR(
         "You need to specify a command.  Use energy --help for details."
      );
   }

   //-------------------------------------------------------------------------
   // Select the ESRV instance to use.
   // Note:
   //    If no ESRV id is provided, then an instance is started and ended
   //    by the program.
   // Note:
   //    If GUID is provided, build the pl_config.ini full path name for the
   //    attach.
   // Note:
   //    If CONFIG is provided, then do nothing, we are ready to attach.
   //-------------------------------------------------------------------------
   memset(
      esrv_config_file_name_buffer,
      0,
      sizeof(esrv_config_file_name_buffer)
   );
   if(p->f_guid == 1) {
      memcpy(
         esrv_config_file_name_buffer,
         ENERGY_ESRV_PL_CONFIG_FILE_ROOT,
         strlen(ENERGY_ESRV_PL_CONFIG_FILE_ROOT)
      );
      strncat(
         esrv_config_file_name_buffer,
         ESRV_APPLICATION_NAME,
         strlen(ESRV_APPLICATION_NAME)
      );
      strncat(
         esrv_config_file_name_buffer,
         ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE,
         strlen(ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE)
      );
      strncat(
         esrv_config_file_name_buffer,
         p->esrv_guid,
         strlen(p->esrv_guid)
      );
      strncat(
```

```
         esrv_config_file_name_buffer,
         ENERGY_ESRV_PL_CONFIG_FILE_NAME,
         strlen(ENERGY_ESRV_PL_CONFIG_FILE_NAME)
      );
      goto attach_to_esrv;
   }
   if(p->f_guid_shell_variable == 1) {
      memcpy(
         esrv_config_file_name_buffer,
         ENERGY_ESRV_PL_CONFIG_FILE_ROOT,
         strlen(ENERGY_ESRV_PL_CONFIG_FILE_ROOT)
      );
      strncat(
         esrv_config_file_name_buffer,
         ESRV_APPLICATION_NAME,
         strlen(ESRV_APPLICATION_NAME)
      );
      strncat(
         esrv_config_file_name_buffer,
         ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE,
         strlen(ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE)
      );
      strncat(
         esrv_config_file_name_buffer,
         p->esrv_guid_shell_variable_value,
         strlen(p->esrv_guid_shell_variable_value)
      );
      strncat(
         esrv_config_file_name_buffer,
         ENERGY_ESRV_PL_CONFIG_FILE_NAME,
         strlen(ENERGY_ESRV_PL_CONFIG_FILE_NAME)
      );
      goto attach_to_esrv;
   }

   //-------------------------------------------------------------------------
   // Because it is easier to type energy -- command than energy --guid xxx --
   // command, we check for the existence of a ENERGY_ESRV_DEFAULT_GUID_SHELL_-
   // -VARIABLE shell variable.  If it exist and has a valid GUID, then it is
   // used.  If no such variable exist, then energy continues by starting its
   // own instance of ESRV.
   //-------------------------------------------------------------------------
   env = getenv(ENERGY_ESRV_DEFAULT_GUID_SHELL_VARIABLE);
   if(env != NULL) {
      ret = plh_filter_uuid_string(env);
      if(ret != PL_FAILURE) {
         memcpy(
            esrv_config_file_name_buffer,
            ENERGY_ESRV_PL_CONFIG_FILE_ROOT,
            strlen(ENERGY_ESRV_PL_CONFIG_FILE_ROOT)
         );
         strncat(
            esrv_config_file_name_buffer,
            ESRV_APPLICATION_NAME,
            strlen(ESRV_APPLICATION_NAME)
         );
         strncat(
            esrv_config_file_name_buffer,
```

```
            ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE,
            strlen(ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE)
         );
         strncat(
            esrv_config_file_name_buffer,
            env,
            strlen(env)
         );
         strncat(
            esrv_config_file_name_buffer,
            ENERGY_ESRV_PL_CONFIG_FILE_NAME,
            strlen(ENERGY_ESRV_PL_CONFIG_FILE_NAME)
         );
         goto attach_to_esrv;
      }
   }

   //------------------------------------------------------------------------
   // Build ESRV binary name and command line
   // Note:
   //   cli_buffer holds the ESRV binary name and the command line options.
   //   if the command and the arguments are provided separately to
   //   CreateProcess then the argv count is erroneous in the started
   //   process and ESRV fails the cli parsing.
   //------------------------------------------------------------------------
   memset(
      esrv_command_buffer,
      0,
      sizeof(esrv_command_buffer)
   );
   strncpy(
      esrv_command_buffer,
      ENERGY_ESRV_BINARY_NAME,
      strlen(ENERGY_ESRV_BINARY_NAME)
   );
   if(p->f_esrv_options == 1) {
      strncat(
         esrv_command_buffer,
         p->esrv_options,
         strlen(p->esrv_options)
      );
   } else {
      strncat(
         esrv_command_buffer,
         ENERGY_ESRV_DEFAULT_OPTIONS,
         strlen(ENERGY_ESRV_DEFAULT_OPTIONS)
      );
   }
   strncat(
      esrv_command_buffer,
      ENERGY_ESRV_SHELL_OPTION,
      strlen(ENERGY_ESRV_SHELL_OPTION)
   );

   //------------------------------------------------------------------------
   // Start an ESRV instance in a child process.
   //------------------------------------------------------------------------
#ifdef __PL_WINDOWS__
```

```
   p->fp_esrv = _popen(
      esrv_command_buffer,
      "rt"
   );
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
   p->fp_esrv = popen(
      esrv_command_buffer,
      "r"
   );
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
   if(p->fp_esrv == NULL) {
      _ERROR("Unable to start ESRV.");
   }

   //-------------------------------------------------------------------------
   // Retrieve the ESRV's instance PID and GUID.
   //-------------------------------------------------------------------------
   do {
      pc = fgets(
         buffer,
         sizeof(buffer),
         p->fp_esrv
      );
      if(pc != NULL) {
         switch(++input_line_count) {

            case ENERGY_ESRV_GUID_LINE:

               //----------------------------------------------------------
               // extract ESRV's GUID and save it
               //----------------------------------------------------------
               token = strtok(
                  buffer,
                  ENERGY_ESRV_GUID_TOKEN_SEPARATORS
               );
               while(token != NULL) {
                  if(strncmp(
                     token,
                     ENERGY_ESRV_PRE_GUID_TOKEN,
                     strlen(ENERGY_ESRV_PRE_GUID_TOKEN)
                  ) == 0) {
                     token = strtok(
                        NULL,
                        ENERGY_ESRV_GUID_TOKEN_TERMINATOR
                     );
                     memset(
                        esrv_guid,
                        0,
                        sizeof(esrv_guid)
                     );
                     strncpy(
                        esrv_guid,
                        token,
                        strlen(token)
                     );
                     f_esrv_guid = 1;
                     break;
```

```
                  }
                  token = strtok(
                     NULL,
                     ENERGY_ESRV_GUID_TOKEN_SEPARATORS
                  );
               }
               break;

            case ENERGY_ESRV_PID_LINE:

               //----------------------------------------------------------
               // extract ESRV's PID and save it.
               //----------------------------------------------------------
               token = strtok(
                  buffer,
                  ENERGY_ESRV_PID_TOKEN_SEPARATORS
               );
               while(token != NULL) {
                  if(strncmp(
                     token,
                     ENERGY_ESRV_PRE_PID_TOKEN,
                     strlen(ENERGY_ESRV_PRE_PID_TOKEN)
                  ) == 0) {
                     token = strtok(
                        NULL,
                        ENERGY_ESRV_PID_TOKEN_TERMINATOR
                     );
#ifdef __PL_WINDOWS__
                     esrv_pid = (DWORD)atoi(token);
#endif // __Pl_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
                     esrv_pid = (pid_t)atoi(token);
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
                     assert(esrv_pid != 0);
                     f_esrv_pid = 1;
                     goto pid_found;
                  }
                  token = strtok(
                     NULL,
                     ENERGY_ESRV_GUID_TOKEN_SEPARATORS
                  );
               }
               break;

            default:
               break;
         }
      } else {

         //----------------------------------------------------------------
         // Likely the ESRV launch has failed, let's signal this error.
         //----------------------------------------------------------------
         _ERROR("ESRV likely failed to start.");
      }
   } while(pc != NULL);

   //----------------------------------------------------------------------
   // Likely the ESRV launch has failed, let's signal this error.
```

```
   //-----------------------------------------------------------------------
   _ERROR("ESRV likely failed to start.");

pid_found:

   //-----------------------------------------------------------------------
   // Check and build the pl_config.ini file to attach to.
   //-----------------------------------------------------------------------
   assert((f_esrv_guid == 1) && (f_esrv_pid == 1));
   memset(
      esrv_config_file_name_buffer,
      0,
      sizeof(esrv_config_file_name_buffer)
   );
   memcpy(
      esrv_config_file_name_buffer,
      ENERGY_ESRV_PL_CONFIG_FILE_ROOT,
      strlen(ENERGY_ESRV_PL_CONFIG_FILE_ROOT)
   );
   strncat(
      esrv_config_file_name_buffer,
      ESRV_APPLICATION_NAME,
      strlen(ESRV_APPLICATION_NAME)
   );
   strncat(
      esrv_config_file_name_buffer,
      ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE,
      strlen(ENERGY_ESRV_PL_CONFIG_FILE_UNDERSCORE)
   );
   strncat(
      esrv_config_file_name_buffer,
      esrv_guid,
      strlen(esrv_guid)
   );
   memset(
      esrv_pl_path_name,
      0,
      sizeof(esrv_pl_path_name)
   );
   strncpy(
      esrv_pl_path_name,
      esrv_config_file_name_buffer,
      strlen(esrv_config_file_name_buffer)
   );
   strncat(
      esrv_config_file_name_buffer,
      ENERGY_ESRV_PL_CONFIG_FILE_NAME,
      strlen(ENERGY_ESRV_PL_CONFIG_FILE_NAME)
   );

attach_to_esrv:

   //-----------------------------------------------------------------------
   // Attach to the identified instance of ESRV and read settings.
   //-----------------------------------------------------------------------
   p->esrv_pld = pl_attach(esrv_config_file_name_buffer);
   if(p->esrv_pld == PL_INVALID_DESCRIPTOR) {
      _ERROR("Unable to attach to the specified ESRV instance.");
```

```
    }

    //---------------------------------------------------------------------
    // read-in esrv's configuration:
    //  - ESRV Status (running or not)
    //  - ESRV Channel count
    //  - ESRV Version (not used)
    //  - ESRV energy in joule counter's precision
    // Note:
    //    since each channel holds esrv's configuration counters, we read the
    //    first channel to read the channel count.  ESRV has always at least
    //    one channel, so the read is safe
    // Note:
    //    Channel count is zero count, therefore the --.
    //---------------------------------------------------------------------
    plret = pl_read(
       p->esrv_pld,
       &p->channels,
       ESRV_COUNTER_CHANNELS_INDEX
    );
    if(plret != PL_SUCCESS) {
       _ERROR("Unable to read the ESRV channels count counter.");
    }
    if(p->channel > p->channels) {
       WARNING(
          "The requested channel does not exist in the specified ESRV instance.  Will
use default channel (1)."
       );
       p->channel = 1;
    }
    p->channel--;

    //---------------------------------------------------------------------
    // Now that the channels count is known, we can read the requested ESRV channel
    //---------------------------------------------------------------------
    plret = pl_read(
       p->esrv_pld,
       &p->status,
       (p->channel * ESRV_BASE_COUNTERS_COUNT) + ESRV_COUNTER_STATUS_INDEX
    );
    if(plret != PL_SUCCESS) {
       _ERROR("Unable to read the ESRV status counter.");
    }
    if(p->status != ESRV_STATUS_RUNNING) {
       _ERROR("The specified ESRV instance doesn't seem to be alive.");
    }
    plret = pl_read(
       p->esrv_pld,
       &p->version,
       (p->channel * ESRV_BASE_COUNTERS_COUNT) + ESRV_COUNTER_VERSION_INDEX
    );
    if(plret != PL_SUCCESS) {
       _ERROR("Unable to read the ESRV version counter.");
    }
    plret = pl_read(
       p->esrv_pld,
       &value,
```

```
      (p->channel * ESRV_BASE_COUNTERS_COUNT) +
ESRV_COUNTER_ENERGY_JOULES_DECIMALS_INDEX
   );
   if(plret != PL_SUCCESS) {
      _ERROR(
         "Unable to read the ESRV energy in Joule(s) counter's .decimal suffix
counter."
      );
   }
   p->energy_data_multiplier = pow(
      10.0,
      (double)value
   );

// do work an use ESRV counters to implement energy efficiency heuristics
...

   //-----------------------------------------------------------------------
   // close the ESRV instance's process and remove its PL
   //-----------------------------------------------------------------------
   if(
      (f_esrv_guid == 1) &&
      (f_esrv_pid == 1)
   ) {
#ifdef __PL_WINDOWS__
      esrv_handle = OpenProcess(
         PROCESS_TERMINATE,
         FALSE,
         esrv_pid
      );
      assert(esrv_handle != NULL);
      bret = TerminateProcess(
         esrv_handle,
         0
      );
      assert(bret != FALSE);

      //--------------------------------------------------------------------
      // reset last ESRV instance's flags and ids
      //--------------------------------------------------------------------
      esrv_handle = NULL;
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
      ret = kill(
         esrv_pid,
         SIGTERM
      );
      assert(ret != -1);
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
      f_esrv_guid = 0;
      f_esrv_pid = 0;
      esrv_pid = 0;

      //--------------------------------------------------------------------
      // close last ESRV instance's output stream
      //--------------------------------------------------------------------
#ifdef __PL_WINDOWS__
      _pclose(p->fp_esrv);
```

```
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
      pclose(p->fp_esrv);
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__

      //--------------------------------------------------------------------
      // Delete ESRV instance's PL.
      //--------------------------------------------------------------------
#ifdef __PL_WINDOWS__
      pc = _getcwd(
         current_path_name,
         sizeof(current_path_name)
      );
      assert(pc != NULL);
      ret = _chdir(esrv_pl_path_name);
      assert(ret != -1);
      file = _findfirst(
         "*",
         &find_files
      );
      do {
         if(
            (
               strcmp(
                  find_files.name,
                  "."
               ) != 0
            ) &&
            (
               strcmp(
                  find_files.name,
                  ".."
               ) != 0
            )
         ) {
            ret = -1;
            do {
               ret = remove(find_files.name);
            } while(ret == -1);
         }
      } while(
         _findnext(
            file,
            &find_files
         ) == 0);
      ret = _findclose(file);
      assert(ret != -1);
      ret = _chdir(current_path_name);
      assert(ret != -1);
      ret = -1;
      do {
         ret = _rmdir(esrv_pl_path_name);
      } while(ret == -1);
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
      directory = opendir(esrv_pl_path_name);
      assert(directory != NULL);
      file = readdir(directory);
```

```
      while(file != NULL) {
         if(
            (
               strcmp(
                  file->d_name,
                  "."
               ) != 0
            ) &&
            (
               strcmp(
                  file->d_name,
                  ".."
               ) != 0
            )
         ) {
            memset(
               file_name,
               0,
               sizeof(file_name)
            );
            strncat(
               file_name,
               esrv_pl_path_name,
               strlen(esrv_pl_path_name)
            );
            strncat(
               file_name,
               "/",
               strlen("/")
            );
            strncat(
               file_name,
               file->d_name,
               strlen(file->d_name)
            );
            ret = -1;
            do {
               ret = unlink(file_name);
            } while(ret != -1);
         }
         file = readdir(directory);
      }
      closedir(directory);
      ret = -1;
      do {
         ret = rmdir(esrv_pl_path_name);
      } while(ret != -1);
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__
   }
done:
   return(PL_SUCCESS);
error:
   return(PL_FAILURE);
}
```

3 TSRV Temperature Monitoring Tool

3.1 TSRV Overview

The Temperature Server (TSRV) is a utility that works with selected temperature and humidity sensors to monitor temperature and humidity. Application developers can integrate TSRV data into their applications to react to changes in temperature or humidity inside or outside servers and network equipment. Additional use cases for Intel EC SDK instrumentation are outlined in the *Intel Energy Checker SDK User Guide*.

As shown in Figure 17 below, the PL GUI Monitor application shipped with the Intel EC SDK can provide a graphical representation of TSRV data in real time.

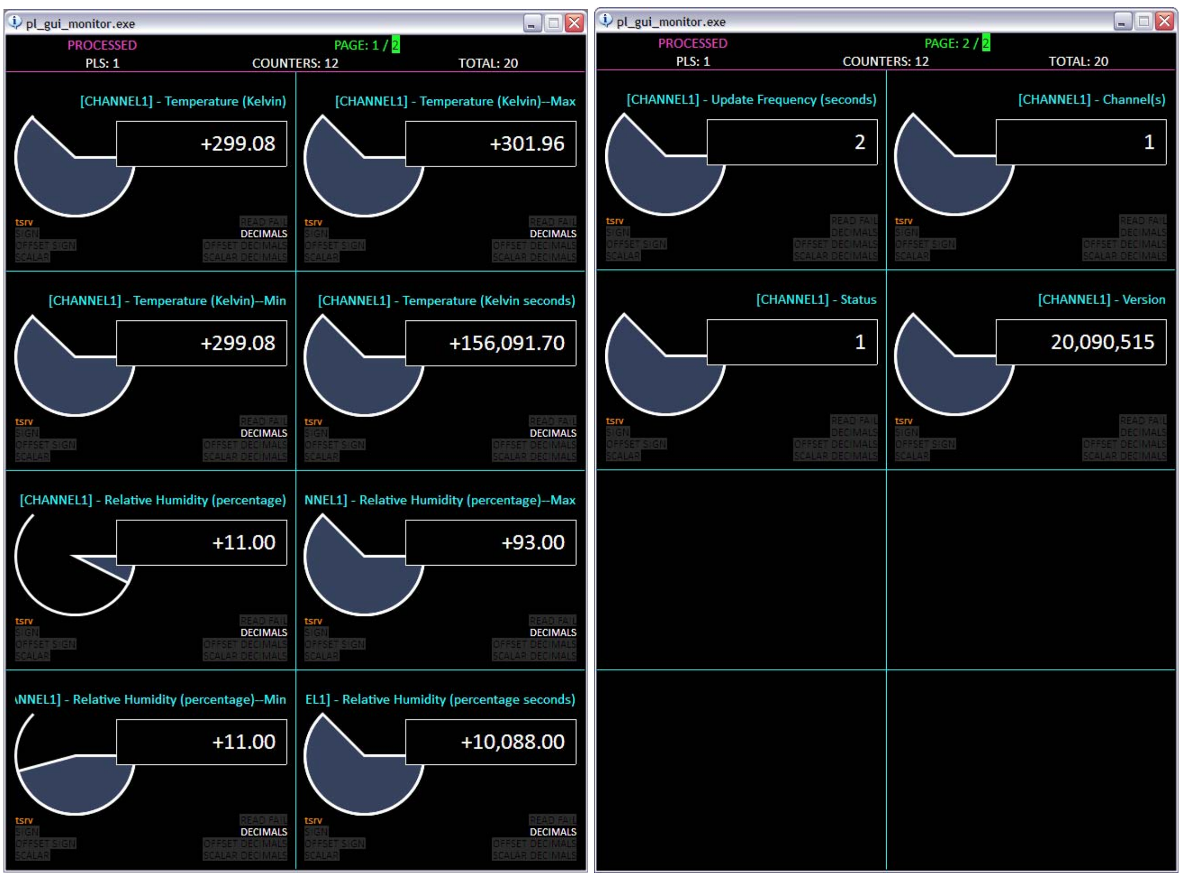

Figure 17: TSRV counters in PL GUI Monitor

3.2 TSRV Hardware Setup

TSRV requires a connection to some device that can measure temperature and/or humidity. This temperature sensor connection could be through a serial port, USB port, locally instrumented temperature sensor, or some other supported interface. The temperature sensor could be local or remote.

NOTE

Multiple instances of TSRV can be hosted by a system, provided that it has the required interfaces and measuring devices in the right quantity. However, when interrupted by the <CTRL>+<C> key combination, all instances of TSRV are stopped.

3.3 TSRV User Interface

TSRV uses a command-line interface similar to the ESRV command-line interface. The user can perform the following operations:

- **start** a TSRV instance
- **stop** one or all TSRV instances on a server
- **reset** the energy counters of one or all TSRV instances on a server

Each command has a contextual help (`--help`).

3.3.1 Starting TSRV

The listing below shows the start command help message: `tsrv --start --help`

```

Start temperature server.

Usage:  tsrv --start --device [dev_name] --device_options[options] --
interface_options[options] [channel] [--diagnostic] [--pause [t]]
Or      tsrv --start --library [lib_name] --device_options[options] --
interface_options[options] [channel] [--diagnostic] [--pause [t]]

   dev_name is one of the following (case insensitive):
      "dh", "Digi_Watchport_H":  Digi* Watchport H external environmental monitoring unit

   lib_name is the filename of a supplemental library for other meters
      The filename is usually a .DLL file in Windows or a .SO file otherwise
      Use quotes around lib_name if it contains spaces

   options (device) refers to the device options (delimited by quotes)
   Check the systems integrator or meter vendor's manual for details

   options (serial) refers to the input interface options (delimited by quotes)
   At the current time, only serial port option strings are supported:
[com=c][baud=b][parity={E|O|N|I}][data={7|8}][stop={0|1|2}][xon={Y|N}][dtr={Y|N}][rts={Y|N}]
      baud={1200|2400|4800|9600|19200|38400|57600|115200|230400}
      For example, "com=0 baud=9600 parity=n data=8 stop=1"
      The default serial options are "com=1 baud=9600 parity=n data=8 stop=1 xon=n dtr=y
rts=n"
```

```

   channel is the interface to monitor on a multi-channel monitoring unit
      For example, to read the second channel on a 3-channel monitoring unit, use 2
      If channel is omitted, channel 1 is implied
      If channel is equal to 0, all channels is implied
      All counters are prefixed with "[CHANNELx] - ", where x is the channel number.

   --diagnostic activates diagnostic messages display during runtime.
   --pause refers to the server's sampling intervale given in seconds.
      by default, pause is 1 second.

   If the temperature server is successfully invoked, then the following elements
   are available:
      A guid printed on the standard output. The guid is a globally unique
      identifier for this set of counters.
      (for example: 273735b1-8319-40fd-b48b-cb9718de415c).
      A "Temperature (Kelvin)" counter.  A Kelvin equals -273.15░C (-457.87░F).
      A "Temperature (Kelvin).decimals" counter (indicating Kelvins is 100x the Kelvin
reading)
      A "Temperature (Kelvin)--Max" counter.
      A "Temperature (Kelvin)--Max.decimals" counter.
      A "Temperature (Kelvin)--Min" counter.
      A "Temperature (Kelvin)--Min.decimals" counter.
      A "Temperature (Kelvin seconds)" counter.
      A "Temperature (Kelvin seconds).decimals" counter.
      A "Relative Humidity (percentage)" counter.
      A "Relative Humidity (percentage).decimals" counter (indicating RH is 100x the Kelvin
reading).
      A "Relative Humidity (percentage)--Max" counter.
      A "Relative Humidity (percentage)--Max.decimals" counter.
      A "Relative Humidity (percentage)--Min" counter.
      A "Relative Humidity (percentage)--Min.decimals" counter.
      A "Relative Humidity (percentage seconds)" counter.
      A "Relative Humidity (percentage seconds).decimals" counter.
      An "Update Frequency (seconds)" counter. Set with the --pause option

   * Third-party trademarks are the property of their respective owners.
```

3.3.1.1. Example Command Lines

Here are examples of how to start TSRV.

```
tsrv --start --device dh
tsrv --start --device dh --interface_options "com=4" --diagnostics
```

NOTE

Under MacOS X, the TSRV `com` *option doesn't take a numerical argument, but takes the name of the cu device. Once the driver is installed, search for its name in the* `/dev` *folder (*`ls -l /dev/cu.*`*). Use the name after the dot (.) for the com option. For example, if the USB-to-serial adapter is named* `/dev/cu.PL2303-000103D`*, then the option is* `com=PL2303-000103D`*.*

3.3.2 Stopping TSRV

The listing below shows the stop command help message: `tsrv --stop --help`

```

Stop temperature server.

Usage:  tsrv --stop [guid] [--diagnostic]

guid is the identification string displayed when tsrv was started.
If guid is 0 or omitted, all active tsrv session are stopped.
--diagnostic activates diagnostic messages display during runtime.
```

3.3.3 Resetting TSRV

The listing below shows the reset command help message: `tsrv --reset --help`

```

Reset temperature and humidity seconds counters to zero.

Usage:  tsrv --reset [guid] [--diagnostic]

guid is the identification string displayed when tsrv was started.
If guid is 0 or omitted, all active tsrv session are reset to zero.
Note: all the channels of the target trsv instance(s) are reset.
--diagnostic activates diagnostic messages display during runtime.
```

3.4 TSRV Integrated Device Support

TSRV has built-in support for the Digi* Watchport/H sensor, which plugs into a USB port (see below).

Figure 18: Digi* Watchport/H sensor

To use this sensor, supply the appropriate device name for the selected temperature sensor. Use --device dh for this sensor, as shown in the example command line below:

```
tsrv --start --device dh
```

3.5 TSRV Library-based Device Support

3.5.1 Simulated Temperature Sensor

Application developers may want to test or demonstrate their application integration when an external temperature or humidity sensor is not available. The Intel EC SDK provides a simulated temperature and humidity sensor that emulates common environmental conditions with a temperature of 298.15 ± 1 kelvin and a relative humidity of 60 ±2 percent.

To use this simulated sensor, use one of the following command lines:

```
tsrv --start --library tsrv_simulated_device.dll (Windows environment)
./tsrv --start --library ./tsrv_simulated_device.so.1.0 (non-Windows environment)
```

3.5.2 TSRV Command-line Library Support

The Intel EC SDK is shipped with a TSRV library that parses output from external applications to read device temperature, hygrometry, and other data via TSRV. In particular, this can be used with IPMItool to query and report appropriate IPMI sensors values as described in section 3.5.3. This command line support can work with virtually any application that prints temperature and relative humidity information to the standard output. In the examples below, a fictitious my_utility application is often used.

NOTE

Refer to section 2.5.10.1 for information on the notation used for command-line library parameters.

3.5.2.1. TSRV Command-line Library Syntax

The TSRV command-line library issues commands to the external tool and reads in the tool's output. If required, the library performs an additional parsing step of the output to isolate the data. Input and output to the tool must be in ASCII format.

The general format for using the TSRV command-line library is as follows:

```
tsrv --start --library ./tsrv_command_line.so.1.0 --device_options <string>
```

The <string> argument includes multiple embedded elements to perform the following actions:

- Specify the tool to use (--tool, --shell)
- Initialize and close the tool (--open_command, --close_command)

- Set the sampling interval (--sampling_interval)
- Characterize the device (--kelvin, --celsius, --fahrenheit)
- Define tokens (--read_*_separators, --read_*_previous_token)
- Read temperature and relative humidity (--read_*_command)

The <string> argument must be in double quotes (“”) since multiple elements must be specified for the TSRV command-line library.

3.5.2.1.1. Specify the Tool to Use

Use the mandatory --tool <string> parameter to identify the application the TSRV command-line library will use to gather temperature data. The argument must be either a fully qualified executable filename or the name of an executable found in the path environment variable. For example, to run my_utility from the /bin directory, include the following within the device_options argument:

```
--tool /bin/my_utility
```

By default, the selected tool will be invoked each time TSRV reads temperature or humidity data. Some tools (i.e., IPMItool) support shell modes. In these modes, the tool only needs to be invoked once, and commands can be repeatedly sent to the same instance of that tool, rather than requiring the TSRV command-line library to repeatedly execute the tool for each command (see section 2.5.10.2.8 for a discussion of performance implications).

To enable shell mode, use the optional --shell flag.

For example, to run my_utility in shell mode, include the following in the device_options argument:

```
--tool my_utility --shell
```

NOTE

The --shell *flag tells the TSRV command-line library to operate in shell mode; it does nothing specifically to put the external tool in shell mode. If a command-line option is needed to start the tool in shell mode, that should be specified in the argument provided in the* --open_command *string (see below).*

3.5.2.1.2. Initialize and Close the Tool

Use the optional --open_command <string> parameter to send any commands to the tool at startup. If the --shell flag is used, this parameter will only be executed once; otherwise, it will be run each time TSRV collects data.

For example, to run my_utility and initialize it with a "quiet mode" command string, include the following within the device_options argument:

```
--tool my_utility --open_command 'quiet mode'
```

NOTE

Since the "quiet mode" string has spaces, it must be quoted. Since it will be placed within the device_options *argument that uses double quotes ("), this command must use single quotes (') to enclose a string.*

Similarly, use the optional --close_command <string> parameter to do any application cleanup before exiting. For example, if the tool needs a bye command to exit, the device_options string might include the following:

```
--tool my_utility --close_command bye
```

NOTE

During system integration and debugging, incorrect parameter exchanges between the TSRV command-line library and the external tool may require the user to manually terminate the external tool each time until the proper shutdown sequence is determined. If the external tool does not terminate properly, TSRV may not terminate.

3.5.2.1.3. Set the Sampling Interval

By default, TSRV will call the data sampling routines at one-second intervals (or at whatever time interval is specified with the --pause command line parameter to TSRV). In rare cases, it may be necessary to increase the interval between samples via the TSRV command-line library. To increase the interval between samples, set the optional --sampling_interval <integer> to a value greater than one. For example, to only sample power with my_utiity at one-third the normal TSRV rate, include the following within the device_options argument:

```
--tool my_utility --sampling_interval 3
```

NOTE

The sampling_interval *is a multiplier of the* pause *parameter. If* sampling_interval *is set to 15 and* pause *is set to 20, TSRV will generate data every 20 seconds but the data will be repeated 15 times. To avoid false interpretations of data, it is generally advisable to omit the command-line library's* sampling_interval *and set the TSRV* pause *parameter to the total desired interval between samples.*

3.5.2.1.4. Characterize the Device

TSRV natively uses kelvin for the temperature scale to keep all temperatures positive. The TSRV command-line library supports three optional command-line flags to accommodate the most common temperature scales:

- --celsius interprets device data as supplied in degrees Celsius
- --fahrenheit interprets device data as supplied in degrees Fahrenheit
- --kelvin interprets device data as supplied in kelvin; since this is the native mode for TSRV, this flag essentially has no impact

NOTE

These device characterization flags only affect TSRV's interpretation of data from the device. TSRV counters are always stored in kelvin, irrespective of the characterization flag provided.

3.5.2.1.5. Define Tokens

The TSRV command-line library issues commands to the external tool and reads the tool's output. Unfortunately, the output from the tool is usually accompanied by some surrounding text and is rarely just a number that can be directly interpreted. For the command-line library to understand how to parse the tool's output, it needs to understand how to delimit tokens in the output stream and how to identify what token in the output stream precedes the value of interest. To support this tokenizing, the TSRV command-line library supports two options for reading the power parameters:

```
--read_temperature_separators <string>
```

```
--read_temperature_previous_token <string>
```

For example, consider the following output from a command line utility used to read system temperature. The values of interest are the value '40' and the word 'Reading'.

```
Sensor ID              : Ambient Temp (0x5)
Sensor Type (Analog)   : Temperature
Sensor Reading         : 40 ( -124) degrees C
Status                 : ok
Lower Non-Recoverable : -10.000
Lower Critical         : 5.000
Lower Non-Critical     : 10.000
Upper Non-Critical     : 50.000
Upper Critical         : 55.000
Upper Non-Recoverable : 65.000
```

In this example, colons and spaces (':' and ' ') delimit tokens. To define the tokens in such an example, include the following within the device_options argument to parse the text above:

```
--read_temperature_separators ': ' --read_temperature_previous_token Reading
```

TSRV offers similar options for humidity:

--read_humidity_separators <string> and --read_current_previous_token <string>

NOTE

The TSRV command-line library does not handle escape codes to specify token separators.

3.5.2.1.6. Read Temperature and Humidity

The TSRV command-line library issues commands to the external tool to retrieve temperature and humidity statistics. For maximum flexibility, each of these commands is independently specified:

- Read temperature: --read_temperature_command <string>
- Read humidity: --read_humidity_command <string>

For example, if my_utility, needs the Retrieve Temp command to determine the current power draw, include the following within the device_options argument:

```
--tool my_utility --read_temperature_command 'Retrieve Temp'
```

Both of these parameters are **mandatory**. Each of the parameters defining commands to read temperature and humidity usually requires corresponding parameters to tokenize the output. For example, if a command line includes the `--read_humidity_command` parameter, it should also include the following parameters:

`--read_humidity_separators` and `--read_humidity_previous_token`

3.5.3 TSRV IPMI Device Support

Many servers manage their power, temperature, and other platform sensors via IPMI, using IPMItool (http://ipmitool.sourceforge.net/). The TSRV command-line library can use IPMItool to query on-board sensors, including sensors for IPMI-instrumented motherboards.

For example, suppose a server has a sensor called `Ambient Temp`. To query this sensor on a system running Linux using IPMItool and the TSRV command-line library, use the following TSRV command line:

```
./tsrv --start --library ./tsrv_command_line.so.1.0 --device_options "--shell --tool
/usr/bin/ipmitool --open_command shell --read_temperature_command 'sensor get
\"Ambient Temp\"' --read_temperature_separators ' :' --read_temperature_previous_token
Reading --celsius" --diagnostic
```

NOTE

For brevity, the commands to read humidity are not shown in the example above.

Note the `read_temperature_command` command in the example above:

`--read_temperature_command 'sensor get \"Ambient Temp\"'`

This is equivalent to the `sensor get "Ambient Temp"` command in IPMItool.

NOTE

IPMI sensor names can vary but the format for reading the sensors will be similar to the example above.

This output of the IPMItool command outlined above is shown below:

```
Sensor ID              : Ambient Temp (0x5)
Sensor Type (Analog)   : Temperature
Sensor Reading         : 40 ( -124) degrees C
Status                 : ok
Lower Non-Recoverable : -10.000
Lower Critical         : 5.000
Lower Non-Critical     : 10.000
Upper Non-Critical     : 50.000
Upper Critical         : 55.000
Upper Non-Recoverable : 65.000
```

The `--read_temmperature_separators ' :' --read_temperature_previous_token Reading` parameters help parse the output stream, as noted in section 3.5.2.1.6 above. The command line also includes the `--celsius` flag, indicating to the TSRV command-line library that the data returned by IPMItool is in celsius, not kelvin.

3.5.4 TSRV Custom Device Libraries

TSRV defines a standard interface to support additional devices beyond those provided with the Intel EC SDK. Vendors wishing to add support for their temperature sensors must provide the appropriate libraries for each supported operating system: a dynamic link library for Windows and shared object files for Linux, Solaris 10, and MacOS X.

Vendors are encouraged to provide support for the entire range of OS types, as well as both 32- and 64-bit versions of TSRV.

The library must implement the seven functions listed below.

```
//-----------------------------------------------------------------------------
// functions prototypes
//-----------------------------------------------------------------------------
TSRV_API int init_device_extra_data(PTSRV);
TSRV_API int delete_device_extra_data(PTSRV);
TSRV_API int open_device(PTSRV, void *);
TSRV_API int close_device(PTSRV, void *);
TSRV_API int parse_device_option_string(PTSRV, void *);
TSRV_API int read_device_temperature(PTSRV, void *, int);
TSRV_API int read_device_humidity(PTSRV, void *, int);
```

NOTE

The `read_device_humidity()` function must still be implemented but can return zero for the humidity value if a temperature sensor does not include a humidity sensor.

3.5.4.1. Code Templates

`\iecsdk\utils\device_driver_kit\src\temperature_meter_driver` is the location in the SDK to find template device code that can be used to build the device library files:

`tsrv_template_device_dynamic_library.c`

`tsrv_template_device_dynamic_library.h`.

The code is heavily commented, and each section of code that may require specific code for a user's device is marked with `// TODO:`.

`\iecsdk\src\temperature_server\pub_tsrv.h` defines a data structure to add the extra data required to manage the device. This data structure is created by TSRV. In the function call flow, TSRV provides two opportunities to initialize the extra data needed to support the device by calling `init_device_extra_data()` function. In between, TSRV calls the `parse_device_option_string()` function, which is an opportunity to update or finalize extra data initialization for the specific device.

NOTE

Dynamically allocated data can be initialized in calls to `init_device_extra_data()`. However, the device library must de-allocate any such resources when TSRV calls the library's `delete_device_extra_data()` function prior to TSRV terminating.

3.5.4.2. Multiple Measurement Channels

TSRV defaults to one measurement channel. If the instrument has more than one measurement channel, set the `virtual_devices` variable to indicate the available number of measurement channels. TSRV supports a maximum of ten channels per measurement device.

If the device supports simultaneous measurement of all the items, and if a `read_device_all_measurements()` function is implemented in the library, then the library should set the `f_optimized_data_read`. Even if the `read_device_all_measurements()` function is found and loaded, it will be called by the TSRV measurement kernel only if the `f_optimized_data_read` flag is set.

Finally, TSRV can also support proprietary interfaces not supported by the base serial TSRV code if `device_interface` is set to `TSRV_DEVICE_INTERFACE_PROPRIETARY` and the appropriate functions to manage that interface are provided; see the documentation in the sample libraries to see how and where this can be done.

The code section below shows how to set the interface flags for a device using a proprietary interface, having three channels and capable of returning all its measurement in a single read access.

```
//-----------------------------------------------------------------------------
// set default virtual device count (channels)
//-----------------------------------------------------------------------------
p->device_data.virtual_devices = 3;

//-----------------------------------------------------------------------------
// signal optimized data measurement capability
//-----------------------------------------------------------------------------
p->f_optimized_data_read = 1;

//-----------------------------------------------------------------------------
// overwrite the default serial interface for library supported devices
//-----------------------------------------------------------------------------
p->device_interface = TSRV_DEVICE_INTERFACE_PROPRIETARY;

```

3.5.4.3. Starting and Closing the Device

Once the device is opened, the TSRV measurement kernel starts to run. At the requested update interval (one second by default), the `read_device_temperature()` and the `read_device_humidity()` functions are called to collect the data. The measurement kernel then exports the data in the PL counters.

TSRV calls the `close_device()` and `delete_device_extra_data()` functions in this order when TSRV is interrupted by the user. The `close_device()` function can execute shutdown code if required by the device, including any code required to manage a proprietary interface. TSRV calls the `delete_device_extra_data()` function to free up dynamically allocated resources prior to program termination.

3.5.4.4. Custom Library Requirements Summary

The listing below summarizes the sequence of functions needed in a library to support a custom device in TSRV.:

```
//----------------------------------------------------------------------------
// function call sequence by the server
//----------------------------------------------------------------------------
// 1 - .....TSRV_API int init_device_extra_data(PTSRV); // first call
// 2 - .....TSRV_API int parse_device_option_string(PTSRV, void *);
// 3 - .....TSRV_API int init_device_extra_data(PTSRV); // second call
// 4 - .....TSRV_API int open_device(PTSRV, void *);
// 5 - N.......TSRV_API int read_device_temperature(PTSRV, void *, int);
//      .......TSRV_API int read_device_humidity(PTSRV, void *, int);
// N + 1 - .TSRV_API int close_device(PTSRV, void *);
// N + 2 - .TSRV_API int delete_device_extra_data(PTSRV);
//----------------------------------------------------------------------------
```

NOTE

The default TSRV update interval is one second between samples. To maximize system performance, this is also the minimum interval between samples supported by TSRV. TSRV has been optimized to reduce performance impacts on the system running TSRV. Developers should be conscious of the overhead their library incurs; excessive processing within the library could negatively affect the adoption of such libraries by TSRV users.

3.6 Using TSRV Counters

TSRV counters can be read through the Intel Energy Checker API, monitored through tools, such as the PL GUI Monitor and PL CSV Logger shipped with the Intel EC SDK, and integrated with tools like the Windows Performance Monitor (Perfmon) application (see Figure 17 and Figure 19).

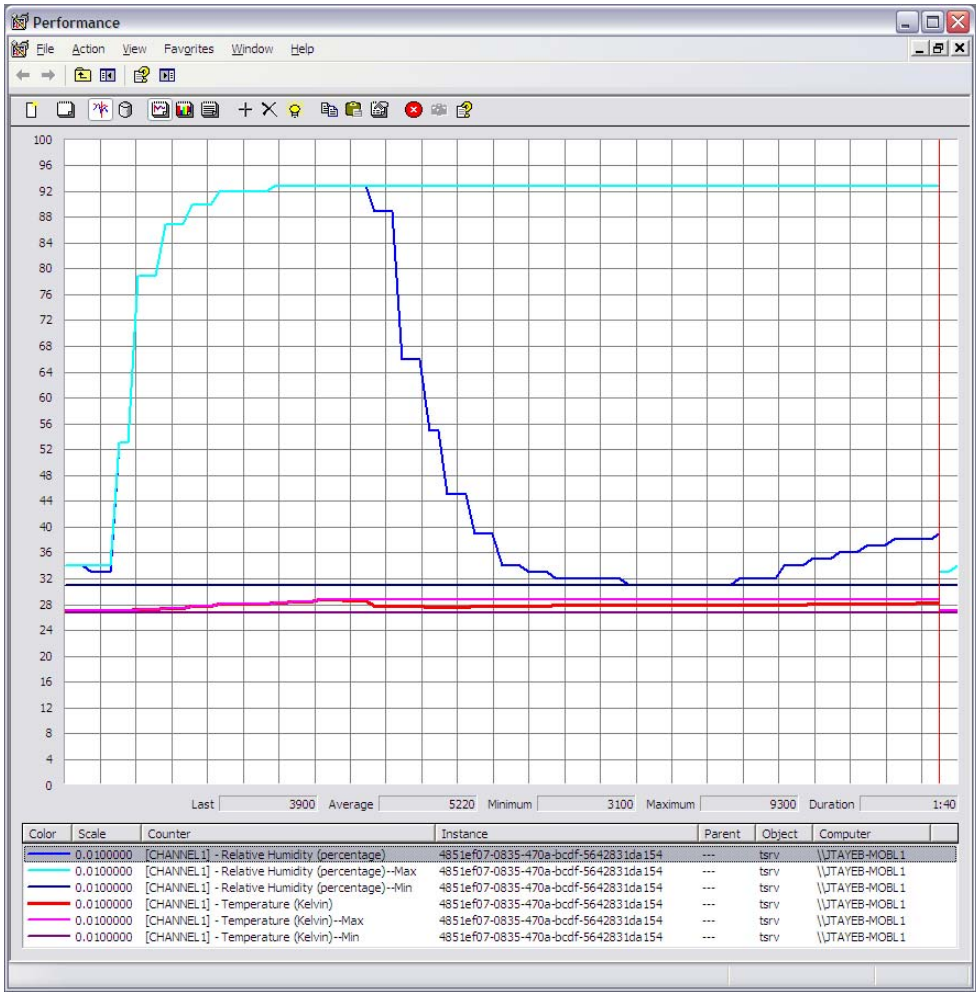

Figure 19: Using pl2w & Perfmon to monitor temperature and humidity

Figure 19 illustrates Perfmon monitoring of TSRV's temperature and relative humidity counters in real time, using the pl2w Windows interoperability tool provided with the Intel EC SDK.

3.6.1 TSRV Exported Counters

TSRV uses the Intel® Energy Checker API to export the counters listed below:

```
1   Temperature (Kelvin)
2   Temperature (Kelvin).decimals
3   Temperature (Kelvin)--Max
4   Temperature (Kelvin)--Max.decimals
5   Temperature (Kelvin)--Min
6   Temperature (Kelvin)--Min.decimals
7   Temperature (Kelvin seconds)
8   Temperature (Kelvin seconds).decimals
9   Relative Humidity (percentage)
10  Relative Humidity (percentage).decimals
11  Relative Humidity (percentage)--Max
12  Relative Humidity (percentage)--Max.decimals
13  Relative Humidity (percentage)--Min
14  Relative Humidity (percentage)--Min.decimals
15  Relative Humidity (percentage seconds)
16  Relative Humidity (percentage seconds).decimals
17  Update Frequency (seconds)
18  Channel(s)
19  Status
20  Version
```

3.6.2 TSRV Counter Descriptions

This section describes all the available TSRV counters. All counters described below are mandatory for supported devices (or zero for devices without a physical sensor corresponding to that counter).

3.6.2.1. Temperature

The **`Temperature (Kelvin)`** counter is provided as a fixed decimal floating-point value. By convention, floating point values are represented by two counter values: the actual counter plus a second counter with a `.decimals` suffix representing the number of digits to the right of the decimal. This additional counter is static. Since the `Temperature (Kelvin).decimals` counter is set to two, this indicates that the `Temperature (Kelvin)` counter is 100 times (10^2 times) the real temperature. For example, a `Temperature (Kelvin)` value of 29815 would actually represent 298.15 Kelvin.

NOTE

Zero degrees Celsius (0°C) is 273.15 Kelvin.

The **`Temperature (Kelvin)--Max`** counter provides the maximum temperature measured since TSRV startup. Since `Temperature (Kelvin)--Max.decimals` is set to two, this indicates that the `Temperature (Kelvin)--Max` counter is 100 times (10^2 times) the real value.

The **`Temperature (Kelvin)--Min`** counter provides the minimum temperature measured since TSRV startup. Since `Temperature (Kelvin)--Min.decimals` is set to two, this indicates that the `Temperature (Kelvin)--Min` counter is 100 times (10^2 times) the real value.

The **`Temperature (Kelvin seconds)`** counter provides the temperature integral over time since TSRV startup. Since `Temperature (Kelvin seconds).decimals` is set to two, this indicates that the `Temperature (Kelvin seconds)` counter is 100 times (10^2 times) the real value.

NOTE

Refer to section 2.6.1.1 for information on integral counters.

3.6.2.2. Humidity

The **`Relative Humidity (percentage)`** counter is provided as a fixed decimal floating-point value. By convention, floating point values are represented by two counter values: the actual counter plus a second counter with a `.decimals` suffix representing the number of digits to the right of the decimal. This additional counter is static. Since the `Relative Humidity (percentage).decimals` counter is set to two, this indicates that the `Relative Humidity (percentage)` counter is 100 times (10^2 times) the real relative humidity (RH). For example, a `Relative Humidity (percentage)` value of 7237 would actually represent 72.37 percent relative humidity.

The **`Relative Humidity (percentage)--Max`** counter provides the maximum RH measured since TSRV startup. Since `Relative Humidity (percentage)--Max.decimals` is set to two, this indicates that the `Relative Humidity (percentage)--Max` counter is 100 times (10^2 times) the real value.

The **`Relative Humidity (percentage)--Min`** counter provides the minimum RH measured since TSRV startup. Since `Relative Humidity (percentage)--Min.decimals` is set to two, this indicates that the `Relative Humidity (percentage)--Min` counter is 100 times (10^2 times) the real value.

The **`Relative Humidity (percentage seconds)`** counter provides the temperature integral over time since TSRV startup. Since the `Relative Humidity (percentage seconds).decimals` counter is set to two, this indicates that the `Relative Humidity (percentage seconds)` counter is 100 times (10^2 times) the real value.

3.6.2.3. Miscellaneous

The **`Update Frequency`** counter provides the number of seconds between samples (1 second by default).

The `channel(s)` counter indicates the number of measurement channel(s) provided by the measurement device used by TSRV.

The `Status` counter indicates if TSRV is running or not. TSRV sets this counter to a non-zero value while running and then resets it to zero when terminating.

Finally, the `Version` counter encodes the TSRV version as its value (TSRV versions are in YYYYMMDD form).

3.6.2.4. Multiple Channels

If multiple channels are supported by the measuring device, then the set of counters is duplicated and appropriately prefixed as shown in section 2.6.1.7

3.6.3 Reading TSRV Counters

This section explains how applications can use TSRV counters to monitor temperature and humidity. More details are provided in the *Intel Energy Checker SDK User Guide*. The goal of this section is to provide developers an overview of how the data they provide may be used from an application.

To facilitate the use of TSRV counters using the Intel EC API, a header file is provided in the SDK (`pub_tsrv_counters.h` in the `\iecsdk\src\temperature_server` folder). This file should be included in each project. The `pub_tsrv_counters.h` file (partially listed below) contains counter index definitions to be used with the `pl_read()` API function calls.

```
//----------------------------------------------------------------------------
// counters definitions.
//----------------------------------------------------------------------------
typedef enum _tsrv_counters_base_indexes {

       TSRV_COUNTER_TEMPERATURE_KELVIN_INDEX = 0,
       TSRV_COUNTER_TEMPERATURE_KELVIN_DECIMALS_INDEX,
       TSRV_COUNTER_MAX_TEMPERATURE_KELVIN_INDEX,
       TSRV_COUNTER_MAX_TEMPERATURE_KELVIN_DECIMALS_INDEX,
       TSRV_COUNTER_MIN_TEMPERATURE_KELVIN_INDEX,
       TSRV_COUNTER_MIN_TEMPERATURE_KELVIN_DECIMALS_INDEX,
       TSRV_COUNTER_TEMPERATURE_KELVIN_SECONDS_INDEX,
       TSRV_COUNTER_TEMPERATURE_KELVIN_SECONDS_DECIMALS_INDEX,
       TSRV_COUNTER_RELATIVE_HUMIDITY_PERCENTAGE_INDEX,
       TSRV_COUNTER_RELATIVE_HUMIDITY_PERCENTAGE_DECIMALS_INDEX,
       TSRV_COUNTER_MAX_RELATIVE_HUMIDITY_PERCENTAGE_INDEX,
       TSRV_COUNTER_MAX_RELATIVE_HUMIDITY_PERCENTAGE_DECIMALS_INDEX,
       TSRV_COUNTER_MIN_RELATIVE_HUMIDITY_PERCENTAGE_INDEX,
       TSRV_COUNTER_MIN_RELATIVE_HUMIDITY_PERCENTAGE_DECIMALS_INDEX,
       TSRV_COUNTER_RELATIVE_HUMIDITY_PERCENTAGE_SECONDS_INDEX,
       TSRV_COUNTER_RELATIVE_HUMIDITY_PERCENTAGE_SECONDS_DECIMALS_INDEX,
       TSRV_COUNTER_UPDATE_FREQUENCY_SECOND_INDEX,

       //------------------------------------------------------------------------
       // tsrv specific counters
       //------------------------------------------------------------------------
       TSRV_COUNTER_CHANNELS_INDEX,
       TSRV_COUNTER_STATUS_INDEX,
       TSRV_COUNTER_VERSION_INDEX

} TSRV_COUNTERS_BASE_INDEXES;

```

3.6.3.1. TSRV Sample Code

Application integration with TSRV is very similar to integration with ESRV. Refer to section 2.6.2 for information on how to integrate applications with ESRV.

The `\iecsdk\utils\device_driver_kit\src\temperature_meter_driver` folder contains `tsrv_template_device_dynamic_library.c` and `tsrv_template_device_dynamic_library.h`, sample code and template libraries, demonstrating how to use TSRV counters and other elements of the Intel EC SDK.