Intel® Energy Efficient Performance Guide

Revision 2.0

August 8, 2013

Revision History

Revision	Revision History	Authors/Contributors	Date
1.0	Initial Release	Jamel Tayeb	2013/x/x
2.0	2013.5.15 Release	Jamel Tayeb	2013/5/15

Abstract

This guide is for software developers who want to understand the energy consumption and performance qualities of their applications. The guide describes how to use the Intel® Software Tester Suite. The guide defines key energy use concepts and presents some sample code. The Intel® Software Tester Suite consists of the Intel® Energy Efficient Performance Tester and a customized API that your application can use to expose performance metrics.

Contents

Figures

Listings

Energy Efficient Performance Guide

1 Introduction

As developers, we should work to improve both the performance and energy consumption of our applications. Our goal is to deliver applications that perform well while consuming the least amount of electrical energy for a given task on a given device. We refer to this as Energy Efficient Performance (EEP).

This guide demonstrates how you can use the Intel® Software Tester Suite to assess the energy efficiency of instrumented and non-instrumented applications. The guide demonstrates this assessment with the iterative optimization of a sample application.

The Intel® Software Tester Suite consists of a module called the Intel® Energy Efficient Performance Tester and a customized API called the Intel® Energy Efficient Performance Tester API that your application can use to expose performance metrics. The Intel® Software Tester Suite is called `esrv` (for energy server) and is sometimes referred to as just the kernel. Currently, the Intel® Software Tester Suite runs under the Windows* operating system.

The sample application discussed in this guide is written in C. You should have Microsoft* Visual Studio with either the Microsoft* C/C++ compiler or the Intel® C/C++ compiler. The sample application works correctly with Visual Studio Express 2012 for Windows Desktop. A number of scripts are provided to aid in running the sample application. These scripts work best when running under the Windows* 8 Operating System.

If you want to install the software and try out an example immediately, go to the section Downloading and Running the Software Tester Suite.

2 The EEP Tester

2.1 Calculating Energy Efficiency

The EEP Tester provides you with the data you need to calculate energy efficiency. The EEP Tester does not report energy efficiency directly.

Define the energy efficiency of a running application as the ratio of the amount of useful work the application performs to the amount of energy consumed by the system when running the application.

$$EE = \frac{Work\ done\ (W)}{Energy\ Consumed\ (E)}$$

Examples of energy efficiency metrics are as follows.

- video frames encoded per Joule
- SQL transactions committed per Joule
- mail messages sent per Joule

To measure the software energy efficiency of your application, you need to measure *both* W (the amount of useful work done) and E (the amount of energy consumed). This guide shows how the EEP Tester can help you take and analyze those measurements.

When you calculate energy efficiency, you can choose one of two methods.

- The first method is to measure only the energy consumed during a fixed task. This method treats the work as a fixed quantity.
- The second method is to measure both the energy consumed and the useful work done: This method requires that you instrument your application so that the EEP Tester captures and correlates work and energy information. This extra effort provides you with finer analysis capabilities, allowing you to put a more targeted optimization plan in place.

Then, modify your application and repeat these measurements. Compare your new measurements with the previous set of measurements. Do this for several versions of your application.

Using the provided scripts ensures that you run your application in an automatic and deterministic way.

The efficiency is the difference in energy consumed when running two different versions of your application.

$$\text{Efficiency} = \Delta E = E1 - E2$$

E1 = Energy consumed for completing Task A in version 1 of your application
E2 = Energy consumed for completing Task A in version 2 of your application

Both methods (measuring only E while keeping W constant and measuring both E and W) require the following steps.

1. Run your application with the EEP Tester. This will likely be your reference run, also called the baseline measurement.

2. Do your performance and/or energy efficiency optimization, creating a new version of your application.
3. Re-run your optimized application with the EEP Tester (the very same way you did in step 1). The EEP Tester provides E and, if you chose the second method, W.
4. Based on the EEP Tester's feedback, reiterate the process at step 2 until you are satisfied.

2.2 Measuring Power

Power draw is measured at the laptop's ACPI battery interface. Power is reported in Watts and is integrated at 1Hz to compute the consumed energy (E) in Joules. Your goal is to reduce the consumed energy for a constant workload.

When the battery is discharging, energy stored chemically in the battery is converted into electrical energy, which in turn is consumed by the system. Hence, the power reading is negative. For example, -35.1 Watts means that the system drew 35.1 Watts from the battery.

All EEP runs should be performed when the laptop is disconnected from wall power with its battery charged to at least 80%. At 80%, your battery should have enough energy to do something useful and not be interrupted because of a low battery condition. In addition, starting at 80% aids reproducibility because the discharge profile of the battery may change based on the charge level. A fully charged battery while connected to the AC power source will have a reading of 0 Watt.

Be sure to use the same power plan for all of your testing. The Balanced Plan provided by a Microsoft* Windows 8 system is a good one to use.

NOTE

> All data collected by the EEP Tester is available in two CSV (comma separated value) files. These are called the raw data files.

Figure 1 shows a typical set of data generated by the EEP Tester loaded in a spreadsheet program (Microsoft* Excel in this case).

- A data file (usually the larger file)
- A key file (will have key in the file name)

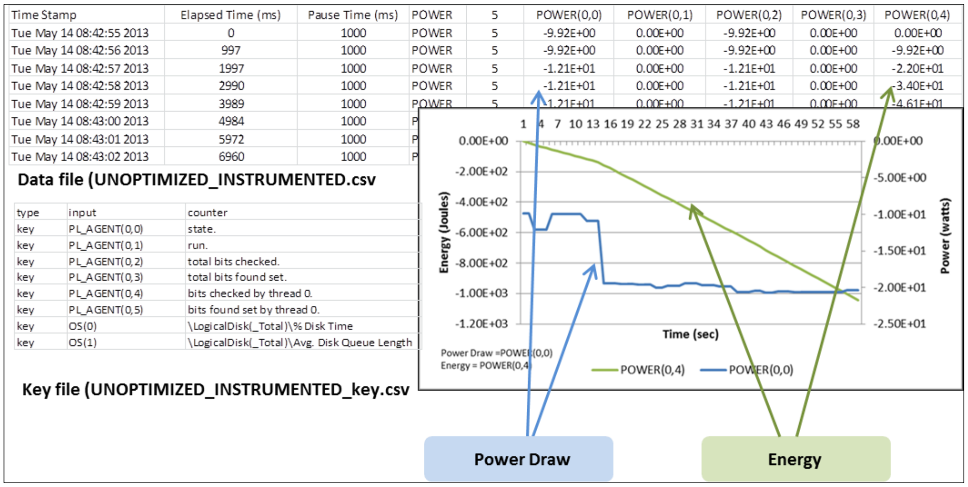


Figure 1: Example of E Data Collected

2.3 Understanding the Data File

Understanding the structure of the data file can help you write your own data tester or to use a spreadsheet or database program more effectively.

The data file contains the raw data collected during a run by the EEP Tester. It is organized in columns and lines. Figure 2 shows a typical data file.

- Each column is a sampled metric of the system or the application. Typical metrics are the total CPU utilization % and the power draw in Watts. A metric is also referred to as a counter.
- Each line is a correlated set of samples for each metric. By default, the sampling interval is approximately one second. The sampling interval is approximate because Windows* is not a hard real-time operating system.

A data file begins with three fixed columns.

- The time stamp is self-explanatory.
- The Pause time (ms) is the sampling interval as requested by the kernel, and it should be constant.
- The Elapsed Time (ms) is the actual elapsed time as measured by the kernel since the beginning of the run.

Fixed fields

A metric group with the metric name POWER

There are 5 metrics in this group: POWER(0,0) through POWER(0,4).

Another metric group called OS with 16 metrics.

Time Stamp	Elapsed Time (ms)	Pause Time (ms)	POWER	5	POWER(0,0)	POWER(0,1)	POWER(0,2)	POWER(0,3)	POWER(0,4)	OS	16	OS(0)
Tue May 14 10:19:44 2013	0	1000	POWER	5	-9.02E+00	0.00E+00	-9.02E+00	0.00E+00	0.00E+00	OS	16	0.00E+00
Tue May 14 10:19:45 2013	993	1000	POWER	5	-9.02E+00	0.00E+00	-9.02E+00	0.00E+00	-9.05E+00	OS	16	0.00E+00
Tue May 14 10:19:46 2013	1981	1000	POWER	5	-9.02E+00	0.00E+00	-9.02E+00	0.00E+00	-1.80E+01	OS	16	0.00E+00
Tue May 14 10:19:47 2013	2982	1000	POWER	5	-1.05E+01	0.00E+00	-1.05E+01	0.00E+00	-2.85E+01	OS	16	6.26E+00
Tue May 14 10:19:48 2013	3982	1000	POWER	5	-1.05E+01	0.00E+00	-1.05E+01	0.00E+00	-3.90E+01	OS	16	0.00E+00

•••

Figure 2: Example of a Data File

After the three fixed columns, a variable number of columns is stored in the data file. Each data line is composed of a variable number of metrics groups. . Figure 2 shows the `POWER` metric group (which contains 5 metrics) and the beginning of the `OS` metric group (which contains 16 metrics).

The structure of each metrics group is parser friendly. That is, a program parsing the data file can find in the file all the information it needs to make sense of the data. The structure of a metrics group is as follows.

- A metric name (also known as an input name).
- A positive integer (the number of metrics in this block. The number of metrics is also known as the inputs or counters count).
- As many indexed metric names as the positive value (as inputs or counters).

A parser, after consuming the three fixed fields of a data file line, can easily interpret the data. The parser can jump over a metrics group and access a given metric in a group.

Figure 3 shows a data file with a `PL_AGENT` metric group. The `PL_AGENT` metrics are user-created. They are created with the EEP Tester API. In your application, you create a metric with `pl_open()`, write it with `pl_write()`, and close it with `pl_close()`. The data file shows an `*` for a metric's value before it is created and an `x` for a metric's value after it is closed. When analyzing data, replace the `*` and the `x` with 0.

PL_AGENT	6	PL_AGENT(0,0)	PL_AGENT(0,1)	PL_AGENT(0,2)
PL_AGENT	6	*	*	*
PL_AGENT	6	*	*	*
			:	
PL_AGENT	6	*	*	*
PL_AGENT	6	*	*	*
PL_AGENT	6	*	*	*
PL_AGENT	6	1	0	0
PL_AGENT	6	1	0	0
PL_AGENT	6	1	1	6336000000
PL_AGENT	6	1	1	9576000000
PL_AGENT	6	1	1	12816000000
			:	
PL_AGENT	6	0	1	1.44E+11
PL_AGENT	6	0	1	1.44E+11
PL_AGENT	6	x	x	x
PL_AGENT	6	x	x	x

••• •••

Figure 3: The `*` and `x` in the `PL_AGENT` Metric Group

Note that a metric's index may be one-dimensional (as in `T(0)`). Some metrics like `PL_AGENT` and `POWER` are two dimensional.

For `PL_AGENT`, the first dimension is the productivity link. A productivity link (PL) is a logical organization of a set of metrics. `PL_AGENT(0,0)` refers to the first metric in the first productivity link. `PL_AGENT(1,0)` refers to the first metric in the second productivity link. For more information about productivity links refer to the section Productivity Links.

For `POWER`, the first dimension is the power measurement channel. With the default configuration (using the ACPI battery interface), only one channel is available. Hence, the value of the first dimension for each metric in the `POWER` group is 0.

Each data file follows the structure just described. Note that the first line of the data file is a header row, but it is structured the same as a data line. Note, however, that the positive integer following the metrics name is guaranteed to be the maximum value found in any subsequent data line.

The metrics count in a group can be dynamic. For example, for a given metrics group, the kernel can collect two metrics during one sample, and five during the next sample. It is important for a parser to record the metrics counts listed in the header as a maximum count.

Figure 4 is an example of this case. The header row (or first data file line) reports a metrics count of 3 for the `T` metrics group. The next data line shows a metrics count of one because that is all that is available at that time. Later three `T` metrics are available, then 2, then none..

T	3	T(0)	T(1)	T(2)
T	1	0	*	*
T	1	0	*	*
T	1	0	*	*
T	1	0	*	*

T	1	4556624	*	*
T	1	4556624	*	*
T	1	4556624	*	*
T	1	4556624	*	*
T	1	4556624	*	*
T	1	4556624	*	*

T	3	27475612	1.79E+09	12315313
T	3	27475612	4.28E+09	12315313
T	3	27475612	6.77E+09	12783034
T	3	27475612	9.27E+09	13196086

T	3	27493677	*	30571177
T	3	27493677	*	30571177
T	3	27493677	*	31351227
T	3	27493677	*	31351227
T	3	*	*	*
T	3	*	*	*

Figure 4: Data File Showing Dynamic Metrics

2.4 Understanding the Key File

The key file describes what the metrics really are. For example, `POWER(0,0)` is the instantaneous power. `POWER(0,4)` is the integrated power or energy. Figure 7 shows an example of a key file.

Figure 5 shows a plot of the `POWER` data. Note the negative readings while the battery is discharged to power the system. The energy (E or `POWER(0,4)` is plotted along a secondary vertical axis (to the right) so that the power draw and E can be viewed on the same graph. Also note the direct relation between the power draw and the E curve's slope. When the power draw increases, the slope of the E curve also increases.

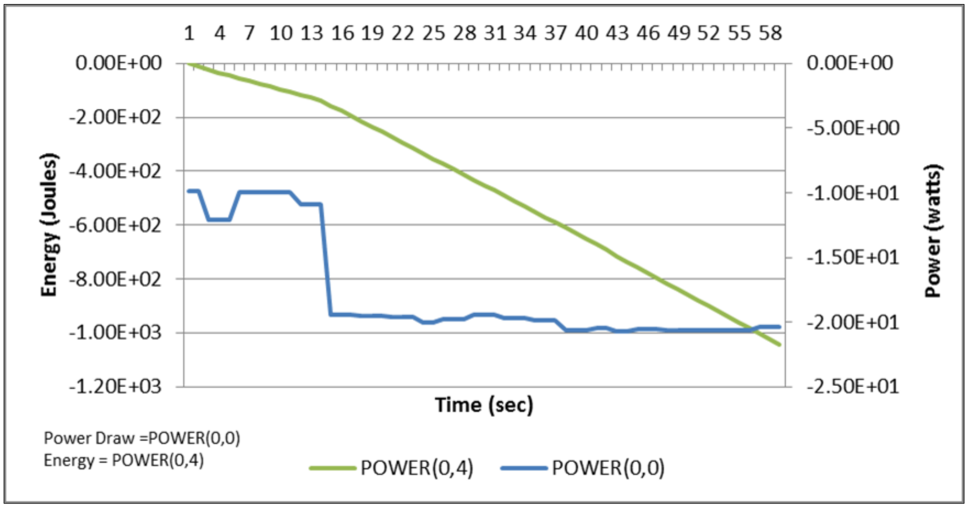


Figure 5: Plot of Average Power Metric Using a Spreadsheet Program

2.5 Start and Stop the Application

Be sure to run your application while the EEP Tester is collecting data.

Using a script ensures that different runs of the same application contain the same input. No checking is performed by the EEP Tester to verify if the application produces the same amount of work for different runs.

2.5.1 Use a State Variable

How do you know when your application is running? You can use the provided API to define and set a state variable. Typically you initialize this variable at 0, set it to 1 when your application starts, and then set it back to 0 when your application ends. The kernel reports this variable as one of the `PL_AGENT` values in the data CSV file.

Figure 6 shows a plot of such a state variable along with the power data.

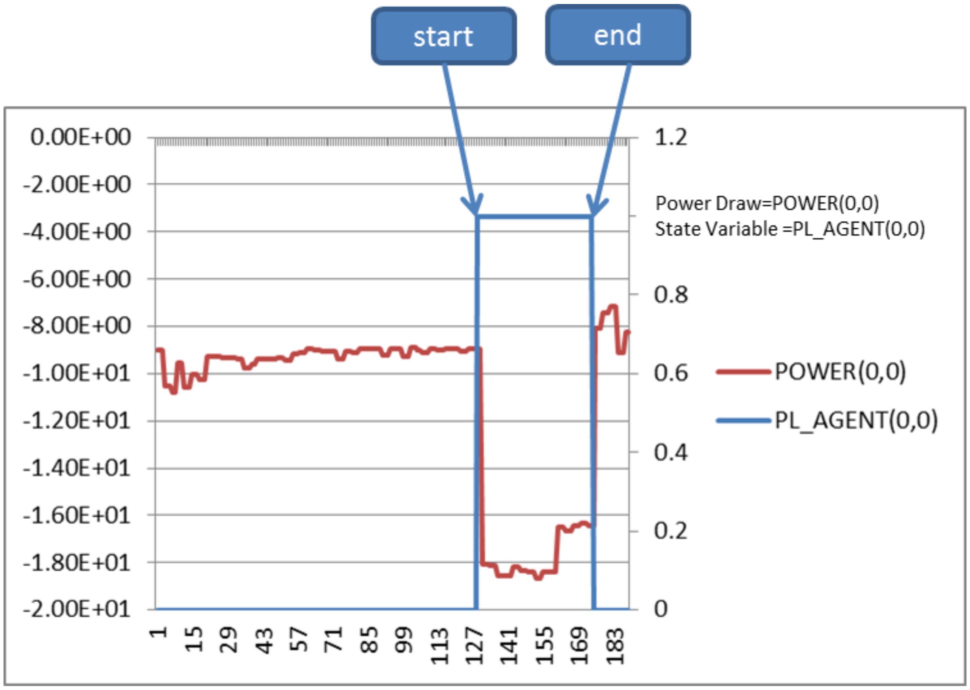


Figure 6: Plot of the Power and a State Variable.

With the provided API, you can also capture and correlate application-specific annotations (application performance data, input related information, etc.).

2.5.2 How to Define a State Variable

Look inside `popcount.c` to see how to set a state variable. `popcount` calls this variable `status`. It is initialized to 0 under the comment `Counters generation variables`. It is set to 1 with `pl_write()` under the comment `Signal start`. It is set back to 0 under the comment `Signal end`, which is under `if(LAST_RUN)`.

3 Instrument the Application

Earlier, we mentioned two measurement methods to collect the data needed to calculate energy efficiency. Refer to the section Calculating Energy Efficiency. The second method was to instrument your application so that it exposed the amount of work done, allowing you to do a more targeted optimization effort. The EEP Tester can capture and correlate this information with power and energy measurements.

Assume that such annotation has been done, and you have collected some data. Figure 7 shows the key file for the data you collected. Note that the annotation data are captured under the `PL_AGENT` metrics group. Figure 8 shows selected data collected by the EEP Tester. In particular, note that the `PL_AGENT(0,0)` and `PL_AGENT(0,2)` metrics are exposed by the application itself using the API.

type	input	counter
key	PL_AGENT(0,0)	state.
key	PL_AGENT(0,1)	run.
key	PL_AGENT(0,2)	total bits checked.
key	PL_AGENT(0,3)	total bits found set.
key	PL_AGENT(0,4)	bits checked by thread 0.
key	PL_AGENT(0,5)	bits found set by thread 0.
key	PL_AGENT(0,6)	bits checked by thread 1.
key	PL_AGENT(0,7)	bits found set by thread 1.
key	PL_AGENT(0,8)	bits checked by thread 2.
key	PL_AGENT(0,9)	bits found set by thread 2.
key	PL_AGENT(0,10)	bits checked by thread 3.
key	PL_AGENT(0,11)	bits found set by thread 3.
key	PL_AGENT(0,12)	bits checked by thread 4.
key	PL_AGENT(0,13)	bits found set by thread 4.
key	PL_AGENT(0,14)	bits checked by thread 5.
	:	
key	OS(14)	\System\Processor Queue Length
key	OS(15)	\System\File Data Operations/sec
key	POWER(0,0)	Power (Watt)
key	POWER(0,1)	Avg Power (Watt) [delta energy / elapsed time]
key	POWER(0,2)	Min Power (Watt)
key	POWER(0,3)	Max Power (Watt)
key	POWER(0,4)	Energy (Joule)
key	T(0)	Dummy Proccess.

Figure 7: Example of the Key File Content Captured by the EEP Tester

`POWER(0,4)` is the energy and it is shown as the blue graph. `POWER(0,4)` measures the power discharge from the battery. `PL_AGENT(0,0)` is the state variable, and it is shown as the green graph. It indicates when `popcount` is running. `PL_AGENT(0,1)` is the total number of bits checked, and it is shown as the red graph. It indicates the total amount of work done.

Figure 8 shows data taken when running `popcount` according to the directions in the section Downloading and Running the Software Tester Suite. Note the use of the State Variable to indicate when `popcount` is running. Note also how the power discharge increases when `popcount` is running. Recall that `popcount` checks a number of bytes for bits set to 1. The work is calculated as the number of bits checked.

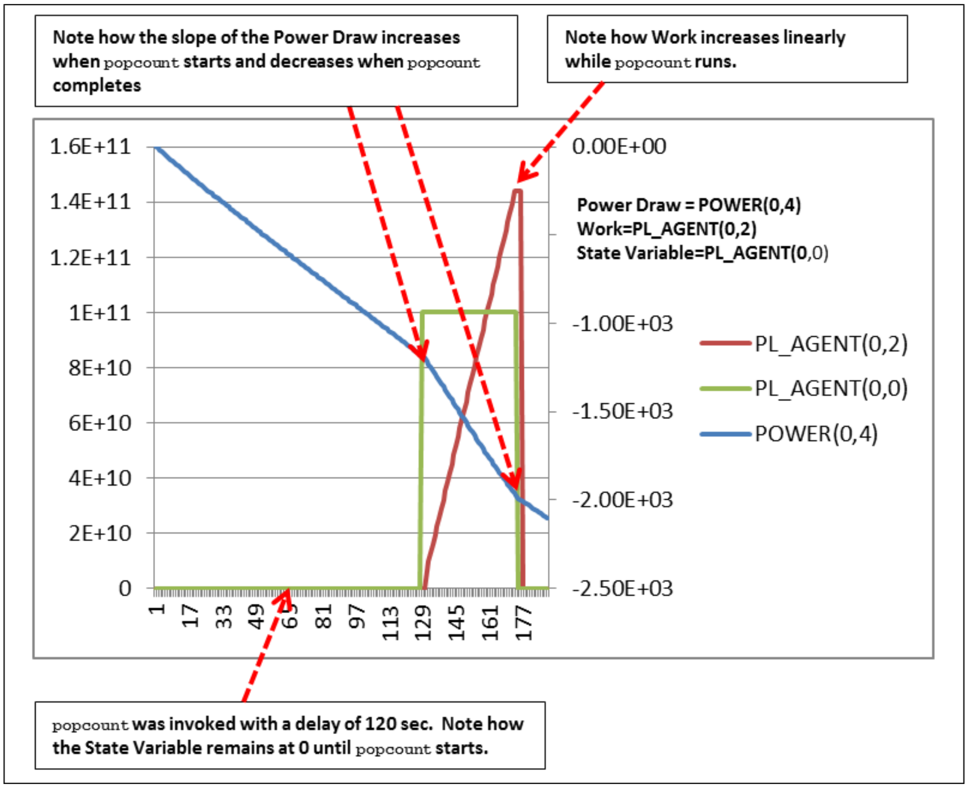


Figure 8: W, E, and Application-Specific Annotation Data

4 Basic API Concepts

4.1.1 Exposing Metrics

To expose any number of metrics from an application, use the provided API. This API has three functions. Using these functions is similar to creating, writing and closing a file:

`pl_open()`	Create an application-specific metrics group.
`pl_close()`	Close an application-specific metrics group created with `pl_open()`.
`pl_write()`	Set the value of a metric of an application specific metrics group created with `pl_open()`.

4.1.2 Counters

From a software developer point of view, a counter is eight bytes of data (64 bits). This matches the `unsigned long long int` C data type. The API provides the functions required for exporting counters from an application.

The specific storage mechanism of the counters is implementation-specific, is not exposed to the applications, and may change in the future.

4.1.3 Counter Export

Exporting a counter means making a counter and its value visible and accessible to other applications. In particular, these counters are visible to the EEP Tester, which can sample them over time. Think of this as writing out the value of the counter. Refer to Figure 9.

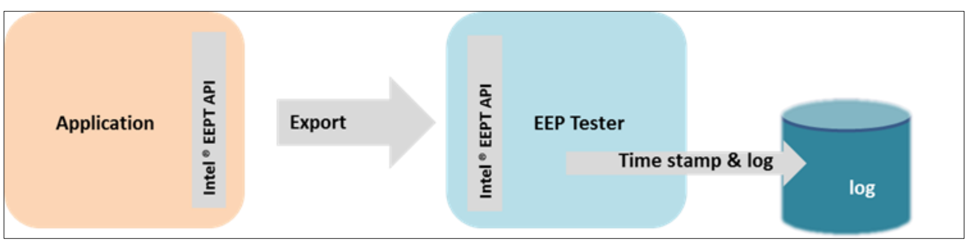


Figure 9: Use of the API to Convey Annotation Data from Application to EEP Tester

4.1.4 Productivity Links

A productivity link (PL) is a logical organization of a set of counters. Imagine a PL as a pipe through which counters are exported. To use a PL to export counters, an application executes the following tasks:

1. Opens a PL.
2. Specifies the counters to be created and managed under the PL.
3. Uses the PL to export counters.
4. At the end of its run, it closes the PL.

An application can have multiple PLs if needed and each PL can have many counters. Refer to Figure 10. Refer to Annotating Threaded Applications for possible and recommended implementations in a multi-threaded application.

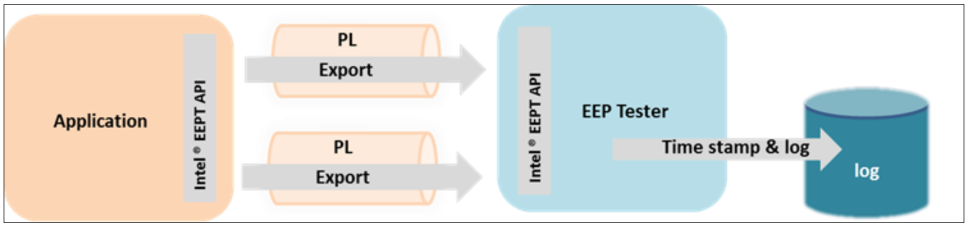


Figure 10: Multiple PL Configuration

NOTE

> The API automatically allows multiple instances of an application to maintain separate metrics for each instance of the application. However, unless your application's architecture is designed for this purpose, do not run multiple instances of your application.

4.1.5 Metric Restrictions and Conventions

4.1.5.1 Storage

PL counters store only numeric data. Inside the application, a counter is an `unsigned long long int` which is eight bytes (64 bits) of memory. Outside the application, counters are stored in a manner determined by the API. Between your application and the EEP Tester, the counters are transmitted via IPC. The API makes the outside nature of the counters transparent for the instrumented applications.

4.1.5.2 Naming

Although the API allows for anonymous counters and applications, anonymous counters should never be used, and counter names should be reasonably descriptive. Recall that counter names are captured in the key file generated by the EEP Tester. The naming conventions for counters are as follows.

- A metric name cannot contain a forbidden character for a file name. Exactly what is forbidden depends on the operating system; but for maximum compatibility, application names and metric names should be chosen from the following set:

  0-9, A-Z, a-z, '_', '-', '(', ')', '[', ']', '.' and the space character

- The length of the metric name plus the application name cannot exceed 199 characters.

NOTE

> Applications are responsible for ensuring that counters meet the naming conventions described above. By default, the API does not check for these conditions. If the `__PL_EXTRA_INPUT_CHECKS__` is defined, then these conditions are checked by the API and the resulting system error is set to `PL_INVALID_PARAMETERS`.

NOTE

> It's good practice to append units (if any) to counter names. For example, if a counter is named `Speed`, rename it to `Speed (mph)`.

4.1.5.3 Ranges

Counters can hold integer values up to 18,446,744,073,709,551,615 (2^64 -1). Most of the events counted can be stored in eight bytes of data. If one of the events being tracked can exceed this maximum number, additional counters can be used to count the overflows (like a carry). To store non-integer values, fixed point notation is used with the decimals suffix as described below.

4.1.5.3.1 Suffix Counters

Sometimes you need to provide additional information about how to interpret the counters. By following certain conventions for counter name suffixes, these counters can be self-describing and can be more easily used by analysis applications. The EEP Tester uses the conventions described in this section to log the actual value of your application's counters.

Since the suffix counters are used to describe the format or meaning of the counters to which they refer, they are static counters (except for the `sign` suffix) and are not expected to be written out more than once per application session. Commonly used suffixes include the following:

```
sign
decimals
offset
offset.decimals
offset.sign
scalar
scalar.decimals
```

NOTE

> The EEP Tester always computes the actual values of your application's counters. It does not distinguish between a static and a dynamic counter.

4.1.5.3.2 Sign Suffix

For maximum portability across different architectures, counters always contain positive values. Negative values can be represented and interpreted by other applications by adhering to the following conventions.

1. Add a supplemental counter with a `.sign` suffix.
2. Write the static sign to the supplemental counter using the following convention.

    1 means a negative number.

    0 means a positive number (or zero).

If the sign suffix is omitted, the number is assumed to be positive. For example, suppose a counter named `height` has a value of 3. If `height.sign` does not exist, or it is zero (0), the composite value of the `height` metric is 3. If `height.sign` has a value of 1, then the composite value of the `height` metric is -3.

NOTE

> The sign suffix counter should be created at the time the base counter is created, if it is expected to represent negative values (even if the current value is positive). If the sign suffix is not defined when the base counter is created, monitoring applications may assume that the value is never expected to be negative.

4.1.5.3.3 Decimals Suffix

Although counters are `unsigned long long int` values, floating point values with a fixed number of decimal places can be represented and interpreted by other applications by adhering to following conventions.

1. Add a supplemental counter with a .decimals suffix.
2. Write the static number of decimal places to the supplemental counter.

For example, consider a counter called `Energy(kWh)`. To represent the value to two decimal places, the actual number of kiloWatt-hours is multiplied by 100 and stored as an `unsigned long long int`. At program startup, write a value of 2 to the `Energy(kWh).decimals` counter to indicate that `Energy(kWh)` has two decimal places. All applications writing counters representing fixed decimal numbers should use a supplemental static `.decimals` suffix counter as appropriate.

4.1.5.3.4 Offset Suffix

In some cases, the native source of a counter may represent a differential value from some fixed offset value. Rather than adding the differential value to the offset value each time the counter is written out, you could elect to define a static offset counter and let any consumers of the data add in the offset when needed. The following convention has been established:

1. Add a supplemental counter with an `.offset` suffix.
2. Write the static value of the offset to the supplemental counter.

For example, if an application is counting the number of visitors over 5000 to a theme park, a `Visitor Total` value of 800 might represent a total attendance of 5800 visitors. By adding a `Visitor Total.offset` counter and writing a value of 5000 to that offset counter, you can add that amount to the counter value to get the real value.

4.1.5.3.5 Scalar Suffix

In most cases, the `decimals` suffix provides sufficient ability to scale the way numbers are reported out. However, there are some cases where the counter values need a scaling factor applied to them. In such cases, the following convention should be used:

1. Add a supplemental counter with a `.scalar` suffix.
2. Write the static value of the scaling factor to the supplemental counter.

For example, consider the software for an egg producer that reports how many dozens of eggs were produced with an `Eggs Dozens` counter. That value could be written to the `Eggs Dozens` counter, provided that `Eggs.scalar` had been set to contain 12.

4.1.5.3.6 Compound Suffixes

The following compound suffixes are commonly recognized:

```
offset.decimals
offset.sign
scalar.decimals
```

For example, consider the following:

The `MyTotal` counter has a value of 5
`MyTotal.scalar` has a value of 72
`MyTotal.scalar.decimals` has a value of 1

In this example, the real scalar value is 7.2 and the actual value is 36 (5 * 7.2). Whenever compound static counters are used, `.decimals` must always be to the right (if used) and `.scalar` must always be to the left of the other static counters if used.

In C notation, the real total for a `total` counter can be figured as shown in Listing 1.

```
real_total =   total * (total.sign ? -1 : 1) / (10 ^ total.decimals)
* total.scalar / (10 ^ total.scalar.decimals)
+ total.offset / (10 ^ total.offset.decimals) * (total.offset.sign ? -1 : 1);
```

Listing 1: The Real Total for a `total` Counter

4.1.5.4 Update Frequency

Applications can use an update frequency that makes sense for that specific application. This frequency can be different for each counter. Some counters will be written only once (for example, the suffix counters described previously). Other counters will be written at regular intervals. For example, the energy counters are updated every second. An update rate higher than once per second is not recommended.

If the granularity of a counter is small, it may be unrealistic to update the associated counter for each counter increment. For example, if the application counts the number of bytes received through a LAN port, then it would be impractical to update the counter for each byte. In this case, updating the counter once every packet or every 100 packets may be a more reasonable approach. The decision of how often to update counters may be handled by the metrics manager thread as indicated in Annotating Threaded Applications.

NOTE

> The EEP Tester uses a dedicated agent to capture the data exported by instrumented applications. All the captured data are collected, and the exact time of transmission and reception is known.
>
> Currently the counters are sampled at the default 1Hz. If you want to capture a state counter's change, you need to keep that counter's value fixed for at least 1s (2s is a safer choice). This limitation does not apply to integral counters (such as the amount of work done) since these are monotonic and increasing.

NOTE

> The API integrates a write cache that cancels write operations if the value to be written is the same as the last one. Even so, it's good practice to avoid writing the same value again and again.

4.1.5.5 Software Status Counter

Another recommended practice is to define a software status counter (`Status`) to indicate when an application is in a given state. For example, an application may use two status values, one to indicate when the application is active and when it is idle. Other applications may have multiple states that can be represented by the same Status counter: 0 = terminated, 1 = idle, 2 = initializing, 3 = active, 4= terminating.

The actual meaning of the `Status` counter is application-specific. However, the presence of the `Status` counter can be useful in performing benchmarking and related analysis activities to determine system idle power consumption overhead. For example, it enables the user to factor out some phases of the algorithm or conversely, to focus on a specific phase. You can use the `Status` counter's values to analyze specified data or regions of interest.

4.1.5.6 Complex Metrics

Counters are the key elements in developing complex metrics. Complex metrics can be built from combinations of counters. An example of a compound metric derived from counters is the amount of work per unit of energy consumed. To use a car analogy, tracking the amount of fuel consumed and the distance traveled can produce a compound metric, commonly expressed as miles per gallon or kilometers per liter.

4.1.6 Annotating Threaded Applications

Although the API is thread-safe, you still should consider some design considerations. Incorporate the API to be as unintrusive as possible and allow it to be turned off easily.

Our recommended approach when instrumenting a threaded application is to create a dedicated metric thread. This thread can be activated on demand (via configuration) and is tasked to collect data from worker threads and expose them using the API. Figure 11 depicts such an organization.

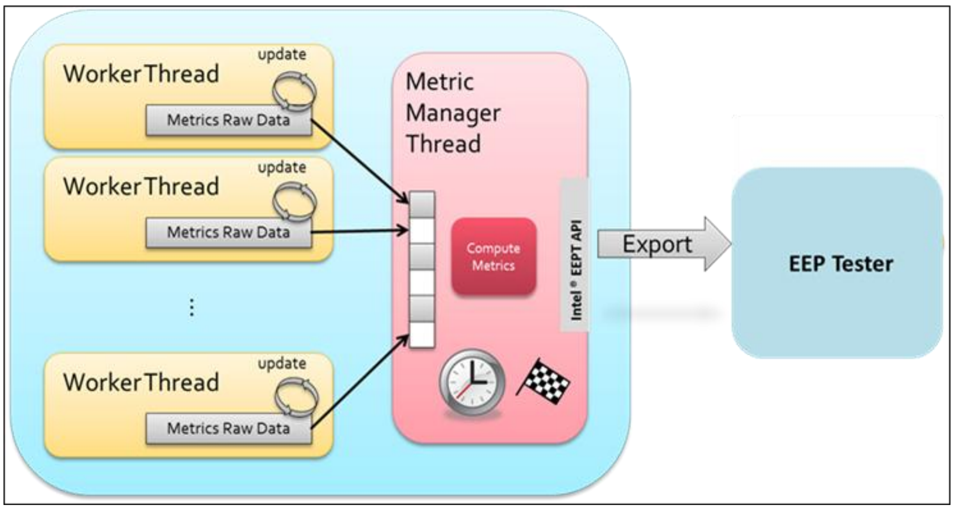


Figure 11: Recommended Instrumentation for Threaded Code

The work unit accounting (updating a counter in the collector/metric manager thread) has to be done within each worker thread. The counters updated by the worker threads could be simple, `unsigned long long int` variables (eight bytes) in an array hosted by the metric manager thread, or they could be sophisticated structures hosted by the metric manager thread and accessed by reference from the worker threads. The decision is up to the application developer.

The actual reporting of the data via API can be adjusted as desired in the metric manager thread, without having to impact the individual worker threads. The metric manager thread can do whatever makes sense from a user point of view, such as never export counters, export them every four hours, export them every second. Even more complex criteria are possible, such as every 20th work item recorded or every 5 minutes, whichever comes first. Of course, the collector thread and the metric(s) computation code (if any) can be aggregated into a single source file.

NOTE

For best results, be sure to align the counter variables to a cache line boundary in memory. This may alleviate a potential performance degradation issue known as “false sharing.” Correct alignment (with the compiler using padding where necessary) resolves this problem.

4.1.7 API Reference

NOTE

IMPORTANT! To work with the EEP Tester, the API code must be configured so that it uses the file system-less mode. This guide only describes software configured for the file system-less mode.

In this mode, the counter data are exchanged via IPC between your application and the EEP Tester.

NOTE

When used with the EEP Tester, the API code must be compiled with the following symbols defined. See Annotating Threaded Applications for build details.

Use these symbols if you do not want to invest time in exploring the API’s configuration options. Options in bold are required to interoperate with the EEP Tester. When reading API detailed information, remember that `__PL_FILESYSTEM_LESS__` should be always defined.

`__PL_WINDOWS__`
`__PL_DYNAMIC_COUNTERS_ALLOCATION__`
`__PL_GENERATE_INI__`
`__PL_GENERATE_INI_VERSION_TAGGING__`
`__PL_GENERATE_INI_BUILD_TAGGING__`
`__PL_GENERATE_INI_DATE_AND_TIME_TAGGING__`
`__PL_BLOCKING_COUNTER_FILE_LOCK__`
`__PL_EXTRA_INPUT_CHECKS__`
`__PL_FILESYSTEM_LESS__`
`__PL_FILESYSTEM_LESS_CONNECTED__`
`__PL_PROFILE__`
`__PL_PROFILE_USE_CONSTANT_FREQUENCY__`
`_WINSOCKAPI_`
`UNICODE`
`_UNICODE`

The API code is provided as a set of two C source code files (`productivity_link.c` and `productivity_link.h`). No run-time software or external libraries are required with the instrumented application; this allows instrumented applications to run standalone, without imposing any additional library dependencies. Alternatively, the API code can be built as Dynamic Link Libraries (DLL) to provide dynamic linkage at runtime.

As shown in Listing 2, an instrumented application should perform the following steps:

1. Create a PL (one call to `pl_open()`, generally performed at initialization time).
2. Expose and update at least one metric (one call to `pl_write()` each time the metric has to be updated).
3. Close the PL previously opened (one call to `pl_close()`, generally at application shutdown).

```
#include <assert.h>
#include "productivity_link.h"

int main(void) {
        //-------------------------------------------------------------------------
        // Generic variables.
        //-------------------------------------------------------------------------
        int ret = PL_FAILURE;
        int pld = PL_INVALID_DESCRIPTOR;
        //-------------------------------------------------------------------------
        // Metric generation variables.
        //-------------------------------------------------------------------------
        unsigned long long int my_value = 42;
        //-------------------------------------------------------------------------
        // PL handling variables.
        //-------------------------------------------------------------------------
        uuid_t my_uuid = { 0 };
        char my_app_name[] = { "My App" };
        char *my_counters[1] = { { "My metric" } };
        //-------------------------------------------------------------------------
        // Open the PL.
        //-------------------------------------------------------------------------
        pld = pl_open(
                my_app_name,
                1,
                my_counters,
                &my_uuid
        );
        assert(pld != PL_INVALID_DESCRIPTOR);
        //-------------------------------------------------------------------------
        // Write the answer to the question.
        //-------------------------------------------------------------------------
        ret = pl_write(
                pld,
                &my_value,
                0
        );
        assert(ret != PL_FAILURE);

        //-------------------------------------------------------------------------
        // Close the PL.
        //-------------------------------------------------------------------------
        Sleep(5000); // <--- this is to let PL_AGENT to capture a nice trace!
        ret = pl_close(pld);
        assert(ret != PL_FAILURE);

        return(0);
}
```

Listing 2: Example of an Instrumented C Program

4.1.7.1 pl_open()

Creates a productivity link (PL) and defines a set of counters in the PL.

Syntax

int pl_open(

```
   char *application_name,
   unsigned int counter_count,
   const char *counter_names[],
   uuid_t *uuid
);
```

Parameters

`application_name`	Pointer to a zero terminated ASCII string.
`counter_count`	Number of counters to create.
`counter_names`	Array of pointers to zero terminated ASCII strings.
`uuid`	Pointer to a `uuid`.

Description

This function is declared in `productivity_link.h`. The function opens a PL and creates `counter_count` counters as specified by the `counter_names` array's entries. The function returns the PL descriptor to be used by all subsequent operations on this PL. The function also writes the `uuid` associated with this PL into the memory location pointed to by `uuid`. It is the caller's responsibility to ensure that the memory location has enough space to hold the `uuid` (`sizeof(uuid_t)`).

Although not recommended, the `application_name` and the `counter_names` array entries can be `NULL`. If `application_name` is `NULL`, then the "`anonymous_application`" string is used instead. If any of the counter_names array's entry is `NULL`, then the "`anonymous_counter_`" string is used instead. For an anonymous counter, its rank is appended to the "`anonymous_counter_`" string.

For example, if `counter_names[0]` is `NULL`, then "`anonymous_counter_1`" is used. If `counter_names[41]` is `NULL`, then "`anonymous_counter_42`" is used, and so on. However, for the sake of clarity, the use of anonymous counters is strongly discouraged.

The `uuid` is returned to the caller for information purposes only. The instrumented application does not need the `uuid` to use the PL.

Sample Code

```
#include <stdio.h>
 #include <windows.h> // only non-PL related system calls are OS specific
 #include <assert.h>
 #include "productivity_link.h"

 //--------------------------------------------------------------------------
 // defines
```

```
//---------------------------------------------------------------------------
#define UPDATE_INTERVAL_IN_MS 1000 // 1 second
#define APPLICATION_NAME "My_Application"
#define COUNTERS_COUNT 2
#define COUNTERS_NAMES {
    "Frames",
    "Pixels"
}
enum COUNTERS {
    FRAMES = 0,
    PIXELS
};

//---------------------------------------------------------------------------
// function prototype
//---------------------------------------------------------------------------
BOOL signal_handler(DWORD);

//---------------------------------------------------------------------------
// program global -- for clarity only
//---------------------------------------------------------------------------
int pld = PL_INVALID_DESCRIPTOR;

//---------------------------------------------------------------------------
// program entry point
//---------------------------------------------------------------------------
int main(void) {

    PL_STATUS ret = PL_FAILURE;
    uuid_t uuid;
    BOOL bret = FALSE;

    char *counters[MAX_COUNTERS] = COUNTERS_NAMES;

    unsigned long long int frames = 0;
    unsigned long long int pixels = 0;

    //-----------------------------------------------------------------------
    // install the event handler routine
    //-----------------------------------------------------------------------
    bret = SetConsoleCtrlHandler(
        (PHANDLER_ROUTINE)signal_handler,
        TRUE
    );
     assert(bret);

    //-----------------------------------------------------------------------
    // open a Productivity Link
    //-----------------------------------------------------------------------
    pld = pl_open(
        APPLICATION_NAME,
        COUNTERS_COUNT,
        counters,
        &uuid
    );
    assert(pld != PL_INVALID_DESCRIPTOR);

    //-----------------------------------------------------------------------
    // direction on how to stop monitoring.
    //-----------------------------------------------------------------------
    fprintf(
        stdout,
        "Type <CTRL>+<C> to stop.\n"
    );

    //-----------------------------------------------------------------------
    // loop until interrupted by user
    //-----------------------------------------------------------------------
    while(TRUE) {
        Sleep(UPDATE_INTERVAL_IN_MS);

```

```
        //------------------------------------------------------------------
        // read meter data
        //------------------------------------------------------------------
        frames = get_encoded_frames_count ();
        pixels = Frames * pixels_per_frame;

        //------------------------------------------------------------------
        // write updated counter in the Productivity Link
        //------------------------------------------------------------------
        ret = pl_write(
            pld,
            &frames,
            FRAMES
        );
        assert(ret == PL_SUCCESS);
        ret = pl_write(
            pld,
            &pixels,
            PIXELS
        );
        assert(ret == PL_SUCCESS);
    }

    //----------------------------------------------------------------------
    // housekeeping is done in the event controller
    //----------------------------------------------------------------------
    return(0);
}

//--------------------------------------------------------------------------
// event handler
//--------------------------------------------------------------------------
BOOL signal_handler(DWORD c) {

    PL_STATUS ret = 0;

    switch(c) {

        case CTRL_C_EVENT:

            //--------------------------------------------------------------
            // process user requested abort
            //--------------------------------------------------------------
            fprintf(stdout, "Stopping...\n");

            //--------------------------------------------------------------
            // close Productivity Link -- global pld comes handy here
            //--------------------------------------------------------------
            ret = pl_close(pld);
            assert(ret == PL_SUCCESS);
            return(FALSE);

         default:
            return(FALSE);
     }
}
```

Listing 3: Sample Code Using `pl_open()`

Return Values

`PL_INVALID_DESCRIPTOR`

Indicates an error condition. If `PL_INVALID_DESCRIPTOR` is returned, then the system's last error code is set as described in Error Codes and Internal Error Codes. Any other return value is a valid PL descriptor.

Error Codes

PL_BYPASSED

The call to pl_open() was bypassed. This happens when the instrumentation is de-activated at compilation time. This is performed by defining the __PL_BYPASS__ symbol.

PL_INVALID_PARAMETERS

At least one argument provided is invalid. This happens if the mandatory pointers are NULL, or if the number of counters is lower than 1 or greater than PL_MAX_COUNTERS_PER_LINK (512 by default). The later check is not performed if the __PL_DYNAMIC_COUNTERS__ symbol is defined. If the __PL_EXTRA_INPUT_CHECKS__ symbol is defined, then additional checks are performed on user inputs as listed below. If any of these checks fail, then this error is returned. It is strongly recommended that this symbol be defined in general, especially if security is a concern.

- Application and counters names are null terminated
- Application and counters names contain only allowed characters. Allowed characters are: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '_', '-', '(', ')', '.', ' ', '[' and ']'
- Application and counters names lengths are within the allowed ranges.
- When built in file system-less mode (when the __PL_FILE_SYSTEM_LESS__ symbol is defined), the following checks are performed:
  - IPV4 address is well formed (length and composition).
  - IPV4 address belongs to Class A, B, C, D or E.
  - Port number is a valid port number.

PL_MISSING_DIRECTORY

The PL_FOLDER doesn't exist. Contact the system administrator to create the PL_FOLDER with the appropriate read and write permissions.

PL_NOT_A_DIRECTORY

PL_FOLDER does exist but is not a directory. Contact the system administrator to create the PL_FOLDER with the appropriate read and write permissions.

PL_DIRECTORY_ALREADY_EXISTS

The directory to be used by the PL already exists. This is a collision case that may happen when a uuid was not guaranteed to be unique. Check the PL_NON_GLOBAL_UUID_DESCRIPTOR and PL_NON_GLOBAL_UUID_DESCRIPTOR_NO_ADDRESS internal error codes for more details.

PL_DIRECTORY_CREATION_FAILED

The directory to be used by the PL cannot be created. Contact the system administrator to check the application's access credentials to the PL_FOLDER.

PL_COUNTER_CREATION_FAILED

The file to be used by a counter cannot be created. Contact the system administrator to check the application's access credentials to the PL_FOLDER.

PL_OUT_OF_MEMORY

There is not enough memory available to allocate data storage. This error may be reported when the counter or the table allocation is performed dynamically. Try freeing up system memory.

PL_FILESYSTEM_LESS_INVALID_IPV4_ADDRESS

An invalid and /or malformed IPV4 address was specified via the PL_AGENT_ADDRESS environment variable. Check the validity and the class of the IPV4 address. This error is returned if any of the following checks fails. This error can be reported only if the __PL_FILESYSTEM_LESS__ symbol is defined.

- IPV4 address is well formed (length and composition)
- IPV4 address belongs to Class A, B, C, D or E

PL_FILESYSTEM_LESS_INVALID_PORT

An invalid port number was specified via the PL_AGENT_PL_PORT environment variable. Check the validity of the port. This error can be reported only if the __PL_FILESYSTEM_LESS__ symbol is defined.

Internal Error Codes

PL_DESCRIPTOR_TABLE_FULL

There is no room to open the PL. The maximum number of PLs is defined by PL_MAX_PRODUCTIVITY_LINKS (ten by default).

PL_NON_GLOBAL_UUID_DESCRIPTOR

The uuid returned and used is not guaranteed to be unique. A collision may occur.

PL_NON_GLOBAL_UUID_DESCRIPTOR_NO_ADDRESS

The uuid generation process is missing a network address. A collision may occur.

PL_GLOBAL_UUID_DESCRIPTOR_CREATION_FAILED

No uuid was generated. This is a critical error.

PL_GLOBAL_UUID_DESCRIPTOR_TO_STRING_FAILED

It was not possible to convert the uuid into a string. This error may happen on very low memory conditions.

PL_PATH_NOT_FOUND

The directory to be used by the PL was not found. This may happen if the directory was deleted by an external application or a user.

PL_COUNTER_SEMAPHORE_CREATION_FAILED

An internal synchronization error happened. Please return a test case reproducing this error to Intel Corporation.

PL_CONFIG_FILE_GENERATION_FAILED

The configuration file (pl_config.ini as defined by PL_CONFIG_FILE_NAME) cannot be created. Either the application's access credentials to the PL_FOLDER are not high enough, or the storage space is limited.

Contact the system administrator to check both conditions. The `pl_config.ini` file is not used n file system-less mode.

`PL_SYNCHRONIZATION_FAILED`
: An internal synchronization error happened. Please return a test case reproducing this error to Intel Corporation.

`PL_CRITICAL_FAILURE`
: An internal synchronization error happened. Please return a test case reproducing this error to Intel Corporation.

`PL_OUT_OF_BUFFER_SPACE`
: An internal buffer ran out of space. This error is not related to the `PL_OUT_OF_MEMORY` error and cannot be solved by freeing up system memory. Please return a test case reproducing this error to Intel Corporation.

`PL_FILESYSTEM_LESS_INITIALIZATION_FAILED`
: An internal error happened while the networking subsystem was initialized. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. Please return a test case reproducing this error to Intel Corporation.

`PL_FILESYSTEM_LESS_SOCKET_FAILED`
: An internal error happened while a socket was created. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_CLOSE_SOCKET_FAILED`
: An internal error happened while a socket was closed. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_CONNECTION_FAILED`
: An internal error happened while a connection to a socket was attempted. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_SEND_FAILED`
: An internal error happened while data were sent through a socket. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_RECV_FAILED`

An internal error happened while data were received from a socket. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_REMOTE_CRITICAL_FAILURE`

A critical error happened on the agent or server side. A critical error may have multiple causes. This error code is returned when the PL protocol is not followed and a bogus message encoding is detected. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

4.1.7.2 pl_close()

Closes a previously opened productivity link (PL) and releases the associated resources.

Syntax

```
int pl_close(
   int pl_descriptor
);
```

Parameters

`pl_descriptor`	A valid productivity link descriptor.

Description

This function is declared in `productivity_link.h`. The function closes the PL identified by the `pl_descriptor` argument and frees up any memory or other resources associated with that PL. The function returns a status code.

Return Values

`PL_SUCCESS`
: Indicates a successful operation.

`PL_FAILURE`
: Indicates an error condition. If `PL_FAILURE` is returned, then the system's last error code is set as described in Error Codes and Internal Error Codes.

Error Codes

`PL_BYPASSED`
: The call to `pl_close()` was bypassed. This happens when the instrumentation is de-activated at compilation time. This is performed by defining `__PL_BYPASS__`.

`PL_DESCRIPTOR_TABLE_UNINITIALIZED`
: The close operation was performed before a successful `pl_open()` call. Open a productivity link before attempting to close one.

`PL_INVALID_PARAMETERS`
: The `pl_descriptor` argument provided is invalid. This happens if the productivity link descriptor is lower than 1 or bigger than `PL_MAX_PRODUCTIVITY_ LINKS` (ten by default).

Internal Error Codes

`PL_SYNCHRONIZATION_FAILED`
: An internal synchronization error happened. Please return a test case reproducing this error to Intel Corporation.

`PL_CRITICAL_FAILURE`
: An internal synchronization related critical error happened. Please return a test case reproducing this error to Intel Corporation.

`PL_FILESYSTEM_LESS_SOCKET_FAILED`

An internal error happened while a socket was created. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_CLOSE_SOCKET_FAILED`

An internal error happened while a socket was closed. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_CONNECTION_FAILED`

An internal error happened while a connection to a socket was attempted. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_SEND_FAILED`

An internal error happened while data were sent through a socket. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_RECV_FAILED`

An internal error happened while data were received from a socket. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_REMOTE_CRITICAL_FAILURE`

A critical error happened on the agent or server side. A critical error can have multiple causes. This error code is returned when the PL protocol is not followed and a bogus message encoding is detected. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

4.1.7.3 pl_write()

Writes a value into a productivity link (PL) counter.

Syntax

```
int pl_write(
   int pl_descriptor,
   const void *pointer_to_data,
   unsigned int counter_offset
);
```

Parameters

`pl_descriptor`	A valid productivity link descriptor.
`pointer_to_data`	A valid pointer to a memory location storing an `unsigned long long int` value.
`counter_offset`	A valid index in the PL's counters list (zero-relative).

Description

This function is declared in `productivity_link.h`. The function writes the value of the variable at `pointer_to_data` into the PL counter identified by `counter_offset`. The variable is an `unsigned long long int` value. `pl_descriptor` identifies the PL to be used.

For example, if the target PL is opened using the counter arguments as shown in the code snippet below, then a subsequent `counter_offset` of 0 writes the data into the `Frames` counter. A `counter_offset` of 1 writes the data into the `Pixels` counter. Listing 4 shows some code that opens a target PL.

```
#define COUNTERS_COUNT 2
#define COUNTERS_NAMES { "Frames", "Pixels" }
const char *counters[COUNTERS_COUNT] = COUNTERS_NAMES;
```

Listing 4: Opening a Target PL

NOTE

> The write operation is cached. This means that no write operation occurs if the value to be written is identical to the previously written value.
>
> The API initializes counters in its cache with a value of 2^64-2. An initial value of 2^64-2 will not be written out to the persistent counter location unless some other value (such as zero) is written to that counter first.

Sample Code

```
#include <assert.h>
#include "productivity_link.h"

//--------------------------------------------------------------------------
// defines
//--------------------------------------------------------------------------
enum COUNTERS { FRAMES = 0 };

unsigned long long int frames = 0;
PL_STATUS ret = PL_FAILURE;

//--------------------------------------------------------------------------
// write frame data
//--------------------------------------------------------------------------
frames = get_encoded_frames_count();
ret = pl_write(
    pld,
    &frames,
    FRAMES
);
assert(ret == PL_SUCCESS);
...
```

Listing 5: Sample Code Using `pl_write()`

Return Values

`PL_SUCCESS`
: Indicates a successful operation.

`PL_FAILURE`
: Indicates an error condition. If `PL_FAILURE` is returned, then the system's last error code is set as described in Error Codes and Internal Error Codes.

Error Codes

`PL_BYPASSED`
: The call to `pl_write()` was bypassed. This happens when the instrumentation is de-activated at compilation time. This is performed by defining `__PL_BYPASS__`.

`PL_DESCRIPTOR_TABLE_UNINITIALIZED`
: The write operation was performed before a successful `pl_open()` call. Open a productivity link before attempting to close one.

`PL_INVALID_PARAMETERS`
: At least one of the arguments provided is invalid. This happens if the productivity link descriptor is less than 1 or greater than `PL_MAX_PRODUCTIVITY_LINKS` (ten by default). This also happens when the destination pointer is `NULL`.

Internal Error Codes

`PL_SYNCHRONIZATION_FAILED`
: An internal synchronization error happened. Please return a test case reproducing this error to Intel Corporation.

PL_COUNTER_FILE_LOCK_FAILED

An internal synchronization error happened. Please return a test case reproducing this error to Intel Corporation.

PL_COUNTER_FILE_ALREADY_LOCKED

An internal synchronization error happened. Please return a test case reproducing this error to Intel Corporation.

PL_COUNTER_FILE_UNLOCK_FAILED

An internal synchronization error happened. Please return a test case reproducing this error to Intel Corporation.

PL_COUNTER_FILE_RESET_FILE_POINTER_FAILED

An internal I/O error happened. Please return a test case reproducing this error to Intel Corporation.

PL_COUNTER_WRITE_FAILED

An internal I/O error happened. Please return a test case reproducing this error to Intel Corporation.

PL_FILESYSTEM_LESS_SOCKET_FAILED

An internal error happened while a socket was created. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

PL_FILESYSTEM_LESS_CLOSE_SOCKET_FAILED

An internal error happened while a socket was closed. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

PL_FILESYSTEM_LESS_CONNECTION_FAILED

An internal error happened while a connection to a socket was attempted. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

PL_FILESYSTEM_LESS_SEND_FAILED

An internal error happened while data were sent through a socket. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

PL_FILESYSTEM_LESS_RECV_FAILED

An internal error happened while data were received from a socket. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

`PL_FILESYSTEM_LESS_REMOTE_CRITICAL_FAILURE`

A critical error happened on the agent or server side. A critical error can have multiple causes. This error code is returned when the PL protocol is not followed and a bogus message encoding is detected. Check if the agent or server is functional and strictly follows the PL protocol. This error can be reported only if the `__PL_FILESYSTEM_LESS__` symbol is defined. If the agent or the server can be excluded as a root cause of the error, please return a test case reproducing this error along with a binary of the agent or server to Intel Corporation.

4.1.8 Building a DLL

Some languages require building a shared object (DLL under Windows) to call the API functions. In addition, one can opt for a dynamic linking of the API. Only the two files (`productivity_link.c` and `productivity_link.h`) are required. The following symbols are required when building the DLL so that the annotation data can be captured by the EEP Tester:

```
__PL_WINDOWS__
_USRDLL
_WINDLL
_UNICODE
UNICODE
__PL_GENERATE_INI__
__PL_GENERATE_INI_VERSION_TAGGING__
__PL_GENERATE_INI_BUILD_TAGGING__
__PL_GENERATE_INI_DATE_AND_TIME_TAGGING__
__PL_BLOCKING_COUNTER_FILE_LOCK__
__PL_DYNAMIC_TABLE_ALLOCATION__
__PL_EXTRA_INPUT_CHECKS__
__PL_FILESYSTEM_LESS__
__PL_FILESYSTEM_LESS_CONNECTED__
__PL_PROFILE__
__PL_PROFILE_USE_CONSTANT_FREQUENCY__
_WINSOCKAPI_
//__PL_JNI_EXPORTS__ add when building a DLL to be used from Java
__PL_WINDOWS_DLL_EXPORTS__
```

NOTE

> By default, the DLLs are compiled with `cdecl` linkage. Some languages such as Microsoft Visual Basic require `stdcall` linkage. In this case, update the project's linker setting appropriately.

4.1.9 API Build Configuration Symbols

The API is provided as source code and uses several symbols to adapt its behavior.

4.1.9.1 Generic Build Configuration Symbols

`_DEBUG`

Define this symbol when building a debug version of the code. This symbol activates the compilation of specific debug code.

`UNICODE` and `_UNICODE`

Define both these symbols when compiling code for Microsoft* Windows operating systems. It is possible to define these symbols solely for the Core API source files.

`__PL_DYNAMIC_COUNTERS_ALLOCATION__`

Define this symbol to activate the dynamic allocation of PL counter data. If this symbol is not defined, then the default counter count limitation (512 counters per

PL with 10 PLs maximum) applies to the code. It is recommended to define this symbol.

4.1.9.2 OS Build Configuration Symbols

`__PL_WINDOWS__`
`_WINSOCKAPI`

Define this symbol when building for Microsoft* Windows operating systems.

4.1.9.3 Dynamic Library Build Configuration Symbols

`__PL_WINDOWS_DLL_EXPORTS__`
`_USRDLL`
`_WINDLL`

Define these symbols when building a dynamic library for Microsoft* Windows operating systems. These symbols also apply to JNI dynamic library generation.

NOTE

> By default, the DLLs are compiled with `cdecl` linkage. Some languages such as Microsoft* Visual Basic require `stdcall` linkage. In this case, update the project's linker setting appropriately.

`__PL_JNI_EXPORTS__`

Define this symbol when building a dynamic library to be used via the Java Native Interface (JNI).

`__PL_LITTLE_ENDIAN__`

Define this symbol when compiling a JNI dynamic library out of the Core API source code files. This symbol is required since the endianess of Java virtual machines (always big endian) may differ from the target platform's processor endianess. This symbol is used when returning the PLs' `UUID` bytes to the JVM.

4.1.9.4 PL Functional Build Configuration Symbols

`__PL_BYPASS__`

Define this symbol to de-activate the Intel® Energy Efficient Performance Tester API functions. When defined, each core API function returns an error code and the system's last error code is set to `PL_BYPASSED`. Applications should gracefully handle this non-error code.

`__PL_GENERATE_INI__`

This symbol is mandatory..

`__PL_GENERATE_INI_VERSION_TAGGING__`

This symbol is mandatory.

`__PL_GENERATE_INI_BUILD_TAGGING__`

This symbol is mandatory.

`__PL_GENERATE_INI_DATE_AND_TIME_TAGGING__`

This symbol is mandatory.

`__PL_BLOCKING_COUNTER_FILE_LOCK__`

This symbol is mandatory. When defined, the access to counter data are synchronized.

`__PL_EXTRA_INPUT_CHECKS__`

This symbol is mandatory. When defined, the following extra checks are performed on the input. Input can be function arguments, environment variables or PL configuration file content.

- Application name contains only allowed characters
- Application name length matches length restrictions
- `UUID` is a well formed `UUID` (length and composition)
- Counter names contain only allowed characters
- Counter names lengths match length restrictions
- IPV4 address is well formed (length and composition)
- IPV4 address belongs to Class A, B, C, D or E
- Port number is a valid port number
- PL protocol encoding is respected

`__PL_FILESYSTEM_LESS__`
`__PL_FILESYSTEM_LESS_CONNECTED__`
`__PL_PROFILE__`
`__PL_PROFILE_USE_CONSTANT_FREQUENCY__`

Define these symbols to activate the file system-less mode of the Intel® Energy Efficient Performance Tester API functions. Refer to the section File System-Less Mode for more details on this mode.

NOTE

> When using Microsoft Visual Studio to build the code under Windows, a summary listing is printed when compiling `productivity_link.c`. Listing 6 is an example of such output.

```
1>productivity_link.c
1>//-----------------------------------------------------------------------
1>// PL Build configuration report.
1>//-----------------------------------------------------------------------
1>NOTE: Building using _DEBUG.
1>NOTE: Building using _UNICODE.
1>NOTE: Building using UNICODE.
1>NOTE: Building using __PL_DYNAMIC_COUNTERS_ALLOCATION__.
1>NOTE: Building using __PL_FILESYSTEM_LESS__.
1>NOTE: Building using __PL_EXTRA_INPUT_CHECKS__.
1>NOTE: Building using __PL_WINDOWS__.
1>NOTE: Building using __PL_GENERATE_INI__.
1>NOTE: Building using __PL_GENERATE_INI_VERSION_TAGGING__.
1>NOTE: Building using __PL_GENERATE_INI_BUILD_TAGGING__.
1>NOTE: Building using __PL_GENERATE_INI_DATE_AND_TIME_TAGGING__.
1>NOTE: Building using __PL_BLOCKING_COUNTER_FILE_LOCK__.
```

Listing 6: Summary Listing Printed when Compiling `productivity_link.c`

4.1.10 File System-Less Mode

The API uses a local or a distributed file system to store the counter data by default. However, some devices, such as mobile phones, tablet PCs or any file system-less embedded system, may not have access to a file system. For these extreme cases, the API can be compiled to run in file system-less mode. To do so, simply define the `__PL_FILESYSTEM_LESS__` symbol during the build process. This will turn the instrumented application into a TCP/IP V4 client, using the PL protocol to communicate with at least one reachable network agent. Agents are the servers in this scenario. Note that the EEP Tester embeds such an agent in its core. This is how annotation data are captured by the tester. For your information, you can refer to the appendix for details on the PL Protocol. Understanding the protocol and the agent configuration is not required to use the EEP Tester.

4.1.11 Interface Examples

NOTE

> Many interface examples implement all API functions. Yet, keep in mind that when built to interoperate with the EEP Tester, the only functions . available to applications are the following.
>
> ```
> pl_open()
> pl_close()
> pl_write()
> ```

4.1.11.1 Using the API in C/C++/Win32

All API description samples are provided in C. Therefore, this section is just a reminder that C/C++ programmers can use the API to implement their application annotation.

4.1.11.2 Using the API in FORTRAN

Listing 7 defines a FORTRAN F90 interface (name the file `productivity_link.f90`). This interface is designed to simplify the use of the PL native functions stored in a dynamic library built from the source code. This sample code can be replaced or amended as needed by the application developer.

```
!------------------------------------------------------------------------------
! Productivity Link FORTRAN interface
!------------------------------------------------------------------------------
module productivity_link

use, intrinsic :: ISO_C_BINDING

implicit none

private
public :: uuid_t, pl_open, pl_close, pl_write

!------------------------------------------------------------------------------
! Types
!------------------------------------------------------------------------------
type uuid_t
  private
  integer(C_SIGNED_CHAR), dimension(16) :: uuid ! Private
```

```
end type uuid_t

!-----------------------------------------------------------------------------
! Interface
!-----------------------------------------------------------------------------
interface

!-----------------------------------------------------------------------------
! Procedures
!-----------------------------------------------------------------------------
  function pl_open (application_name, n_counters, counter_names, uuid) bind(C)
    import
    integer(C_INT) :: pl_open
    character, dimension(*), intent(in) :: application_name
    integer(C_INT), value, intent(in) :: n_counters
    integer(C_INTPTR_T), dimension(*), intent(in) :: counter_names ! Array of pointers
 to strings
    type(uuid_t), intent(out) :: uuid
  end function pl_open

  function pl_attach (config_name) bind(C)
    import
    character, dimension(*), intent(in) :: config_name
  end function pl_attach

  function pl_close (pld) bind(C)
    import
    integer(C_INT) :: pl_close
    integer(C_INT), value, intent(in) :: pld
  end function pl_close

  function pl_read (pld, val, counter_id, counter_offset) bind(C)
    import
    integer(C_INT) :: pl_read
    integer(C_INT), value, intent(in) :: pld
    integer(C_LONG_LONG), intent(in) :: val
    integer(C_INT), value, intent(in) :: counter_id
    integer(C_INT), value, intent(in) :: counter_offset
  end function pl_read

  function pl_write (pld, val, counter_id) bind(C)
    import
    integer(C_INT) :: pl_write
    integer(C_INT), value, intent(in) :: pld
    integer(C_LONG_LONG), intent(in) :: val
    integer(C_INT), value, intent(in) :: counter_id
  end function pl_write

end interface

end module productivity_link

```

Listing 7: A FORTRAN F90 Interface

Listing 8 shows how to use the Intel EC API from FORTRAN F90. The example creates a simple PL with four (4) counters and sets the values of two (2) of them.

```
      !-----------------------------------------------------------------------
      !  fortran_calling_sample.f90
      !  PROGRAM: fortran_calling_sample
      !  PURPOSE:  Entry point for the fortran_calling_sample.
      !-----------------------------------------------------------------------
      program fortran_calling_sample
      use productivity_link
      use, intrinsic :: ISO_C_BINDING
      implicit none
```

```

    integer, parameter :: COUNTERS_COUNT = 4

    integer(C_INT) :: pld, ret
    character(*), parameter :: application_name = "my_fortran_application"//C_NULL_CHAR
    integer(C_INTPTR_T), dimension(COUNTERS_COUNT) :: counter_names
    type(uuid_t) :: uuid
    integer(C_LONG_LONG) :: val1 = 987654321
    integer(C_LONG_LONG) :: val2 = 123456789

    !------------------------------------------------------------------------
    ! Fill in the counter_names
    ! Using some extensions here - more complicated to do with strict F2003 and
    ! it would obscure the code
    !------------------------------------------------------------------------
    counter_names = [LOC("The Amazing A Counter"C), &
                     LOC("The not so bad B Counter"C), &
                     LOC("Counter C"C), &
                     LOC("Counter D"C)]

    !------------------------------------------------------------------------
    ! Create and open a PL
    !------------------------------------------------------------------------
    pld = pl_open (application_name, COUNTERS_COUNT, counter_names, uuid)

    !------------------------------------------------------------------------
    ! Write a couple counters
    !------------------------------------------------------------------------
    ret = pl_write (pld, val1, 0)
    ret = pl_write (pld, val2, 1)

    !------------------------------------------------------------------------
    ! Close the PL
    !------------------------------------------------------------------------
    ! do sleep for 5s // <--- this is to let PL_AGENT to capture a nice trace!
    ret = pl_close (pld)

    end program fortran_calling_sample
```

Listing 8: Using the FORTRAN F90 Interface

4.1.11.3 Using the API in Java

Listing 9 defines a Java* class (name the file ProductivityLink.java). This class is designed to simplify the use of the PL native functions stored in the dynamic library built from the source code. This sample code can be replaced or amended as needed by the application developer.

NOTE

> Do not forget to define the `__PL_JNI_EXPORTS__` symbol when building a DLL to be used by Java codes.

```
//------------------------------------------------------------------------
// Productivity Link JNI interface class
//------------------------------------------------------------------------
import java.util.UUID;

public class ProductivityLink {

   //---------------------------------------------------------------------
   // functions jni interfaces
   //---------------------------------------------------------------------
```

```
   public native int pl_open(String pl_application_name, int pl_counters_count, String
pl_counters_names[], UUID puuid);
   public native int pl_close(int pld);
   public native int pl_write(int pld, Long counter, int counter_index);

   //-----------------------------------------------------------------------
   // enums
   //-----------------------------------------------------------------------
   public enum _pl_status {
      PL_SUCCESS,
      PL_FAILURE;
   }

   public enum _pl_failure {

        PL_INVALID_DESCRIPTOR (0x10000000),
        PL_BYPASSED,
        PL_INVALID_PARAMETERS,
        PL_SYNCHRONIZATION_FAILED,
        PL_MISSING_DIRECTORY,
        PL_NOT_A_DIRECTORY,
        PL_NO_ACCESS,
        PL_DIRECTORY_CREATION_FAILED,
        PL_DIRECTORY_ALREADY_EXISTS,
        PL_PATH_NOT_FOUND,
        PL_DESCRIPTOR_TABLE_FULL,
        PL_DESCRIPTOR_TABLE_UNINITIALIZED,
        PL_NON_GLOBAL_UUID_DESCRIPTOR,
        PL_NON_GLOBAL_UUID_DESCRIPTOR_NO_ADDRESS,
        PL_GLOBAL_UUID_DESCRIPTOR_CREATION_FAILED,
        PL_GLOBAL_UUID_DESCRIPTOR_TO_STRING_FAILED,
        PL_CRITICAL_FAILURE,
        PL_CONFIG_FILE_GENERATION_FAILED,
        PL_CONFIG_FILE_OPENING_FAILED,
        PL_COUNTER_CREATION_FAILED,
        PL_COUNTER_SEMAPHORE_CREATION_FAILED,
        PL_COUNTER_ATTACH_FAILED,
        PL_COUNTER_TO_STRING_FAILED,
        PL_COUNTER_WRITE_FAILED,
        PL_COUNTER_FILE_RESET_FILE_POINTER_FAILED,
        PL_COUNTER_READ_FAILED,
        PL_COUNTER_FILE_LOCK_FAILED,
        PL_COUNTER_FILE_ALREADY_LOCKED,
        PL_COUNTER_FILE_UNLOCK_FAILED,
        PL_COUNTER_VALUE_OUT_OF_RANGE,
        PL_OUT_OF_MEMORY,
        PL_OUT_OF_BUFFER_SPACE,
        PL_BLOCKING_PL_READ_INSTANCE_CREATION_FAILED,
        PL_BLOCKING_PL_READ_INSTANCE_DESTRUCTION_FAILED,
        PL_BLOCKING_PL_READ_HANDLE_CREATION_FAILED,
        PL_BLOCKING_PL_READ_HANDLE_DESTRUCTION_FAILED,
        PL_BLOCKING_PL_READ_HANDLE_RENEWING_FAILED,
        PL_BLOCKING_PL_READ_WAITING_NOTIFICATION_FAILED,
        PL_FILESYSTEM_LESS_REMOTE_CRITICAL_FAILURE,
        PL_FILESYSTEM_LESS_INITIALIZATION_FAILED,
        PL_FILESYSTEM_LESS_NETWORK_ADDRESS_RESOLUTION_FAILED,
        PL_FILESYSTEM_LESS_SOCKET_FAILED,
        PL_FILESYSTEM_LESS_CLOSE_SOCKET_FAILED,
        PL_FILESYSTEM_LESS_CONNECTION_FAILED,
        PL_FILESYSTEM_LESS_SEND_FAILED,
        PL_FILESYSTEM_LESS_RECV_FAILED,
        PL_FILESYSTEM_LESS_INVALID_IPV4_ADDRESS,
        PL_FILESYSTEM_LESS_INVALID_PORT,
        PL_COUNTER_WRITE_CACHE_HIT,
        PL_COUNTER_WRITE_CACHE_MISS,
        PL_NO_ERROR;

      private int failure_code = 0;

      private _pl_failure(int code) {
         this.failure_code = code;
```

```
      }

      private _pl_failure() {
         this.failure_code = 0;
      }
   }

   //--------------------------------------------------------------------------
   // constants definitions
   //--------------------------------------------------------------------------
   public static class _pl_constants
   {
      static final int PL_MAX_PRODUCTIVITY_LINKS = 10;
      static final int PL_MAX_COUNTERS_PER_LINK = 250;
      static final int PL_CONFIGURATION_FILE_APPLICATION_NAME_LINE = 1;
      static final int PL_CONFIGURATION_FILE_UUID_STRING_LINE = 2;
      static final int PL_CONFIGURATION_FILE_LOCATION_LINE = 3;
      static final int PL_CONFIGURATION_FILE_COUNTERS_NUMBER_LINE = 4;
   }

   static {
      System.loadLibrary("productivity_link_jni");
   }
}
```

Listing 9: Productivity Link JNI Interface Class

Listing 10 shows how to use the Intel EC API from Java. The example creates a simple PL with four (4) counters and sets the values of two (2) of them.

```
 import java.util.UUID;
public class ProductivityLinkDemo {
   public static void main(String[] args) {
      int pld;
      String application_name = "my_java_application";
      String counter_names[] = {
         "The Amazing A Counter",
         "The not so bad B Counter",
         "Counter C",
         "Counter D"
      };
      UUID uuid = new UUID(0, 0);
      Long val1 = new Long(987654321);
      Long val2 = new Long(123456789);

      //--------------------------------------------------------------------------
      // create and open a PL
      //--------------------------------------------------------------------------
      ProductivityLink jpl = new ProductivityLink();
      pld = jpl.pl_open(application_name, counter_names.length, counter_names, uuid);

      //--------------------------------------------------------------------------
      // write few counters
      //--------------------------------------------------------------------------
      jpl.pl_write(pld, val1, 0);
      jpl.pl_write(pld, val2, 1);

      //--------------------------------------------------------------------------
      // close the PL
      //--------------------------------------------------------------------------
      Thread.sleep(5000); // <--- this is to let PL_AGENT to capture a nice trace!
      jpl.pl_close(pld);
    }
}
```

Listing 10: Using the Java Interface

4.1.11.4 Using the API in .NET* in C#

Like the JNI interface in Java, .NET* provides the `InteropServices` assembly to interface managed and unmanaged code. The code below defines a C# class (name the file `ProductivityLink.cs`).

Listing 11 shows the use of PL counters from C# code. This code sample assumes that this is a Windows environment, that the DLL is named `productivity_link.dll`, and that DLL is visible to the application's binary at runtime. Figure 12 shows how the annotation data of this sample are captured by the EEP Tester and saved in the key file.

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Threading; // for Sleep

namespace ProductivityLinkDemo {

   class Program {

      //--------------------------------------------------------------------
      // the entry point
      //--------------------------------------------------------------------
      [STAThread]
      static void Main(string[] args) {

         int pld;
         string application_name = "my_CSharp_application";
         string [] counter_names = {
            "The Amazing A Counter",
            "The not so bad B Counter",
            "Counter C",
            "Counter D"
         };
         Guid uuid = Guid.NewGuid();
         ulong val1 = 987654321;
         ulong val2 = 123456789;

         //-----------------------------------------------------------------
         // create and open a PL
         //-----------------------------------------------------------------
         pld = ProductivityLink.pl_open(application_name, counter_names.Length,
counter_names, ref uuid);

         //-----------------------------------------------------------------
         // write few counters
         //-----------------------------------------------------------------
         ProductivityLink.pl_write(pld, ref val1, 0);
         ProductivityLink.pl_write(pld, ref val2, 1);

         //-----------------------------------------------------------------
         // close the PL
         //-----------------------------------------------------------------
         Thread.Sleep(5000); // <--- this is to let PL_AGENT to capture a nice trace!
         ProductivityLink.pl_close(pld);
      }
   }

   public class ProductivityLink {

      //--------------------------------------------------------------------
      // functions InteropServices interfaces
      //--------------------------------------------------------------------
      [DllImport("productivity_link.dll", CharSet = CharSet.Ansi, EntryPoint =
"pl_open", ExactSpelling = false, CallingConvention = CallingConvention.Cdecl)]
      public static extern int pl_open(string pl_application_name, int
pl_counters_count, string[] pl_counters_names, ref Guid puuid);
      [DllImport("productivity_link.dll", CharSet = CharSet.Ansi, EntryPoint =
```

```
"pl_attach", ExactSpelling = false, CallingConvention = CallingConvention.Cdecl)]
      public static extern int pl_attach(string pl_config_file_name);
      [DllImport("productivity_link.dll", EntryPoint = "pl_close", ExactSpelling =
false, CallingConvention = CallingConvention.Cdecl)]
      public static extern int pl_close(int pld);
      [DllImport("productivity_link.dll", EntryPoint = "pl_read", ExactSpelling =
false, CallingConvention = CallingConvention.Cdecl)]
      public static extern int pl_read(int pld, ref ulong counter, int counter_index);
      [DllImport("productivity_link.dll", EntryPoint = "pl_write", ExactSpelling =
false, CallingConvention = CallingConvention.Cdecl)]
      public static extern int pl_write(int pld, ref ulong counter, int
counter_index);
    }}
```

Listing 11: Using PL Counters from C#

type	input	counter
key	PL_AGENT(0,0)	The Amazing A Counter
key	PL_AGENT(0,1)	The not so bad B Counter
key	PL_AGENT(0,2)	Counter C
key	PL_AGENT(0,3)	Counter D

Figure 12: Example of Key File Section Captured for the C# Example

4.1.11.5 Using the API in a scripting language

If your application (or its component implementing the annotation) is not native or managed, but rather a script, you can use SWIG and language-specific C interface facilities. For illustration purposes only (please refer to the SWIG documentation and your scripting language documentation for details), Listing 12 shows a SWIG interface file for Python.

```
//--------------------------------------------------------------------------------
// Note:
//    The following environment variables need to be defined. In addition, the
//    Python binary must be in the PATH.
//    PYTHON_INCLUDE=C:\Python27\include
//    PYTHON_LIB=C:\Python27\libs\python27.lib
// Note:
//    The SWIG generated wrap file for productivity link is created and deleted
//    at each build.
//--------------------------------------------------------------------------------

%module productivity_link

//--------------------------------------------------------------------------------
// Typemaps (uuid_t *, const void *, char **).
//--------------------------------------------------------------------------------
%typemap(in) uuid_t * {
      $1 = (uuid_t *)malloc(sizeof(uuid_t));
}

%typemap(argout) uuid_t * {
      size_t i = 0;
      size_t l = 16;
      size_t size = PyList_Size($input);
      unsigned char *p = (unsigned char *)$1;
      if(16 >= size) {
             l = size;
      }
      for(i = 0; i < l; i++) {
             PyList_SetItem(
                    $input,
```

```
            i,
            PyInt_FromLong((long)p[i])
        );
    }
    for(; i < 16; i++) {
        PyList_Append(
            $input,
            PyInt_FromLong((long)p[i])
        );
    }
    free((uuid_t *)$1);
}

%typemap(in) const void * {
    unsigned long long v = 0;
    if(PyLong_Check($input)) {
        v = PyLong_AsUnsignedLongLong($input);
    } else {
        if(PyInt_Check($input)) {
            v = PyInt_AsUnsignedLongLongMask($input);
        }
    }
    $1 = &v;
}

%typemap(in) char ** {
    if(PyList_Check($input)) {
        size_t size = PyList_Size($input);
        unsigned int i = 0;
        $1 = (char **)malloc((size + 1) * sizeof(char *));
        for(i = 0; i < size; i++) {
            PyObject *o = PyList_GetItem(
                $input,
                i
            );
            if(PyString_Check(o)) {
                $1[i] = PyString_AsString(
                    PyList_GetItem(
                        $input,
                        i
                    )
                );
            } else {
                free($1);
                return(NULL);
            }
        }
        $1[i] = 0;
    } else {
        return(NULL);
    }
}

%typemap(freearg) char ** {
    free((char *) $1);
}

%inline %{

//-----------------------------------------------------------------------------
// Headers.
//-----------------------------------------------------------------------------
#ifdef __PL_WINDOWS__
    #include <windows.h>
#endif // __PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
    #include <uuid/uuid.h>
#endif // __PL_LINUX__ || __PL_SOLARIS__ || __PL_MACOSX__

//-----------------------------------------------------------------------------
// Functions prototypes.
```

```
//-----------------------------------------------------------------------------
#if defined (__PL_FILESYSTEM_LESS__) &&
  defined (__PL_FILESYSTEM_LESS_CONNECTED__) &&
  defined (__PL_PROFILE__)
	extern int pl_open(char *, unsigned int, const char **, uuid_t *);
	extern int pl_close(int);
	extern int pl_write(int, const void *, unsigned int);
#else
// __PL_FILESYSTEM_LESS__ && __PL_FILESYSTEM_LESS_CONNECTED__ && __PL_PROFILE__
	extern int pl_open(char *, unsigned int, const char **, uuid_t *);
	extern int pl_attach(const char *);
	extern int pl_close(int);
	#ifndef __PL_LOGGER_ONLY__
		extern int pl_read(int, void *, unsigned int);
	#endif // __PL_LOGGER_ONLY__
	extern int pl_write(int, const void *, unsigned int);
#endif
// __PL_FILESYSTEM_LESS__ && __PL_FILESYSTEM_LESS_CONNECTED__ && __PL_PROFILE__

%}
```

Listing 12: Sample SWIG Interface File (Python)

5 A Sample EEP Test Application

5.1 Example

To demonstrate the use of the EEP Tester, we have developed a test application that counts the number of bits set in a given byte stream. This operation is also known as `popcount` in computer science (or population count), and is used in various very important fields such as communications or cryptography. We will use this application to demonstrate the following techniques.

- Performance optimization of the code
    - Serial optimizations
    - Parallel optimizations
- Annotation of the code for EEP
    - Measuring EEP With annotation

When optimizing energy efficiency, two distinct but complementary paths can be taken. The first is optimizing for performance. Doing so shortens the execution time and very likely increases the power consumption. This is typically not a problem because energy is the integral of power over time, and the time is less. Often the energy decreases.

The second path is optimizing for pure energy efficiency. For a fixed performance level (the same work done during the same run time), reduce the energy consumption. This is a hard problem with no simple recipe.

A book dedicated to Energy-Aware Computing can be found here:
http://noggin.intel.com/intelpress/categories/books/energy-aware-computing .

5.1.1 `popcount`

`popcount` scans bytes to count the number of bits set. Define useful work as the total number of bits checked. Multiple algorithms, well documented in the literature, can be used to count bits set in a byte. Listing 13 shows a trivial algorithm as our baseline.

```
unsigned long long int trivial_get_popcount(PBYTE p, size_t l) {
        size_t i = 0;
        BYTE b = 0;
        unsigned long long int count = 0;
        assert(p != NULL);
        assert(l > 0);
        for(i = 0; i < l; i++) {
                b = p[i];
                for(; b; b >>= 1) {
                        count += b & 1;
                }
        }
        return(count);
}
```

Listing 13: Trivial Implementation of `popcount` (implicit, default –trivial)

5.1.2 Optimize the code

This section describes only a few of the many possible optimization techniques. Refer to http://software.intel.com/en-us/intel-sdp-home for more information about optimizing code.

A profiler can be used to quickly identify your code's hot spots. For information on the Intel® VTune™ Amplifier XE profiling tool, refer to http://software.intel.com/en-us/intel-vtune-amplifier-xe. For `popcount`, we focus on the `popcount` function because it is the biggest consumer of CPU cycles.

Two paths for the performance optimization of the bit detection function will be examined now: serial optimizations and parallel optimizations.

5.1.3 Optimizing for Serial Performance

Assuming that your code is compiled (either native or managed), start by using an optimizing compiler. To see the optimizing compilers available from Intel, refer to http://software.intel.com/en-us/c-compilers/.

This guide uses the Microsoft* Visual Studio 2010 C/C++ compiler with default Release build settings to generate the binaries. An optimizer compiler directed to generate code for a targeted platform can yield substantial performance gains for a relatively limited effort. Remember that you can selectively compile the hot spot functions with a different compiler (such as the Intel® C/C++ or FORTRAN compilers).

One optimization such a compiler can perform is called loop unrolling. The compiler may also use vector instructions assuming it can identify independent iterations of the data. The compiler may need hints expressed with pragmas.

5.1.3.1 Using Intrinsics

Our first optimization consists of using the compiler intrinsic for the `popcount` operation. Listing 14 shows our implementation of this optimization. Note that the compiler can still be used for additional optimizations with this code.

```
unsigned long long int intrin_get_popcount(PBYTE p, size_t l) {
        size_t i = 0;
        unsigned long long int count = 0;
        assert(p != NULL);
        assert(l > 0);
        for(i = 0; i < l; i++) {
                count += __popcnt(p[i]);
        }
        return(count);
}
```

Listing 14: Implementation of `popcount` Using Compiler Intrinsics (`--intrinsic` option)

5.1.3.2 Using a Lookup Table

Listing 15 shows an optimization that trades performance for memory footprint. Indeed, a look-up table can often be advantageously used to speed-up the computation. With a byte resolution, a lookup table will only use 256 bytes, which can be accommodated by most modern processor data caches.

```
unsigned long long int library_get_popcount(PBYTE p, size_t l) {
        size_t i = 0;
        unsigned long long int count = 0;
        BYTE lookup[] = { LOOKUP_TABLE_DATA };
        assert(p != NULL);
```

```
        assert(l > 0);
        for(i = 0; i < l; i++) {
                count += lookup[p[i]];
        }
        return(count);
}
```

Listing 15: Implementation of `popcount` Using a Lookup Table (`--library` option)

5.1.3.3 Using Hardware Acceleration

We can also use hardware acceleration (if available) for the best possible result. With SSE4.2 in the IA, a `popcount` instruction is available. Becasue compilers may not always spot the potential for using such instructions, we explicitly code the `popcount` SSE instruction in our code, as shown in Listing 16. This explicit coding can be performed either with inline assembly (not recommended) or with compiler intrinsics. Note that this requires some extra caution especially in the data layout. Note also that special care should be taken to accommodate the addressing mode's impact on what data sizes and intrinsics to use.

```
unsigned long long int hardware_get_popcount(PBYTE p, size_t l) {
        size_t i = 0;
        size_t loops_count = 0;
        mm_data_type *px = (mm_data_type *)p;
        unsigned long long int count = 0;
        assert(p != NULL);
        assert(l > 0);
        loops_count =
                (l / sizeof(mm_data_type)) -
                ((l % sizeof(mm_data_type)) == 0)
        ;
        for(i = 0; i <= loops_count; i++) {
                count += _mm_popcnt(*px++);
        }
        return(count);
}
```

Listing 16: Implementation of `popcount` Using Hardware Acceleration (`--hardware`)

We have applied three serial optimizations to `popcount`. These techniques represent a good gradation in both implementation difficulties for the developer and performance gains for the user. Refer to Figure 13.

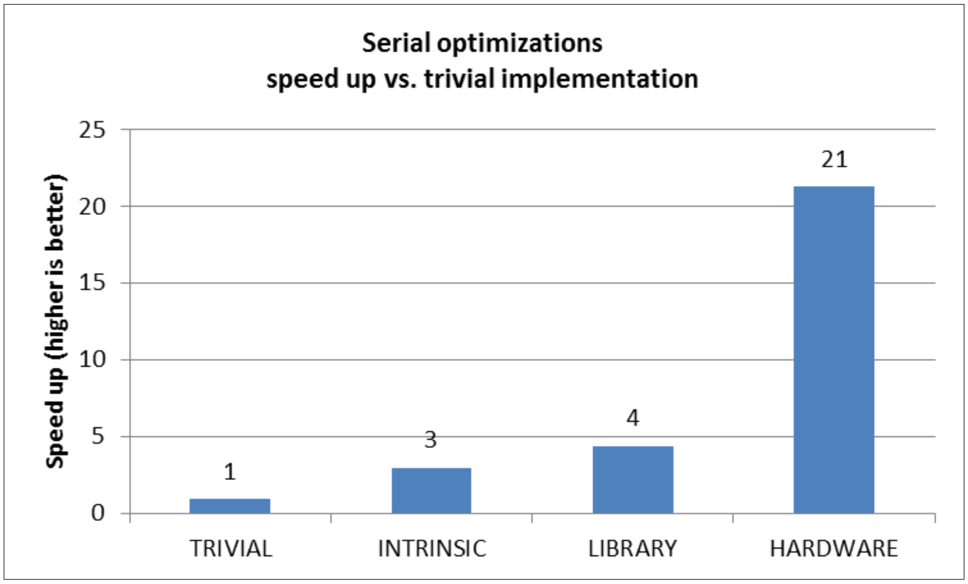


Figure 13: Serial Optimization Speed vs. Trivial Implementation.

5.1.3.4 Optimize Parallel Performance

The second performance optimization path is to leverage the multicore and multithreaded architectures of modern processors. Parallelizing an application is not an easy task. However, if the processing done by the code is well suited for parallelization (functional and/or data parallelism), the gains can be spectacular.

NOTE

> `popcount` was implemented such that serial execution is just a special case of a parallel execution. When `popcount` runs in serial mode, one worker thread is used to process the entire data set.

`popcount` can be parallelized at the fine-grained data level with an optimizing compiler or manually by the programmer. Recall the use of vector instructions mentioned in the serial performance optimization section.

Parallelization can also be introduced at the coarse grain data level by implementing a pool of worker threads sharing the workload. We threaded the code using the Win32 API, but other software libraries would work as well. For more information on Intel software tools, view http://threadingbuildingblocks.org/.

Split the workload among a variable number of threads that then search for and find the set bits. Refer to the source code for implementation details. Keep in mind that such parallelization may require substantial code and data layout changes.

Listing 17 shows the implementation of the worker threads function. Many changes other than those shown in the listing were required, so please refer to the source code for details. Note that for the set bits detection, via the function pointer (at line 41), we run any of the available algorithms (presented in the serial optimizations section). Figure 14 shows the parallel speedup for a 4-core HT-enabled system.

```
unsigned int __stdcall worker_thread_function(void *p) {

        //--------------------------------------------------------------------
        // Generic variables.
        //--------------------------------------------------------------------
        unsigned int i = 0;
        BOOL bret = FALSE;
        size_t rank = 0;

        //--------------------------------------------------------------------
        // timing variables.
        //--------------------------------------------------------------------
        LARGE_INTEGER start_time = { 0 };
        LARGE_INTEGER end_time = { 0 };
        LARGE_INTEGER frequency = { 0 };

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

        //--------------------------------------------------------------------
        // Prepare workload.
        //--------------------------------------------------------------------
        assert(p != NULL);
        rank = *(size_t *)p;
        assert(rank < threads.threads_count);

        //--------------------------------------------------------------------
        // Start thread's run time measurement.
        //--------------------------------------------------------------------
        QueryPerformanceFrequency(&frequency);
        QueryPerformanceCounter(&start_time);

        //--------------------------------------------------------------------
        // The workload!
        //--------------------------------------------------------------------
        if(FIRST_RUN) {
                threads.p_threads_data[rank].thread_counts_data.bits_checked = 0;
                threads.p_threads_data[rank].thread_counts_data.bits_found_set = 0;
        }
        for(i = 0; i < options.iterations; i++) {
                threads.p_threads_data[rank].thread_counts_data.bits_found_set +=
                        threads.p_threads_data[rank].f(
                                &problem_data[threads.p_threads_data[rank].start],
                                threads.p_threads_data[rank].bytes
                        )
                ;
                threads.p_threads_data[rank].thread_counts_data.bits_checked +=
                        (threads.p_threads_data[rank].bytes * BITS_PER_BYTE)
                ;
        }

        //--------------------------------------------------------------------
        // End and report thread's run time measurement (and some extra stats).
        //--------------------------------------------------------------------
        QueryPerformanceCounter(&end_time);
        threads.p_threads_data[rank].run_duration_in_ms = (
                (double)(end_time.QuadPart - start_time.QuadPart) *
                1000.0 / (double)frequency.QuadPart
        );
        printf(
                WORKER_THREAD_REPORT_FORMAT_STRING,
                threads.threads_id_string,
                (unsigned int)threads.runs + 1,
                (unsigned int)rank,
                (unsigned int)threads.p_threads_data[rank].start,
                (unsigned int)threads.p_threads_data[rank].end,
                (unsigned int)(
                        threads.p_threads_data[rank].end -
```

```
                        threads.p_threads_data[rank].start +
                        1
                ),
                (unsigned int)threads.p_threads_data[rank].bytes,
                threads.p_threads_data[rank].thread_counts_data.bits_checked,
                threads.p_threads_data[rank].thread_counts_data.bits_found_set,
                threads.p_threads_data[rank].run_duration_in_ms / 1000.0
        );

        //-------------------------------------------------------------------
        // Signal worker thread's completion to driver.
        //-------------------------------------------------------------------
        assert(threads.p_events[rank] != NULL);
        bret = SetEvent(threads.p_events[rank]);
        assert(bret == TRUE);

        return(PL_SUCCESS);
}
```

Listing 17: Implementation of the Worker Threads

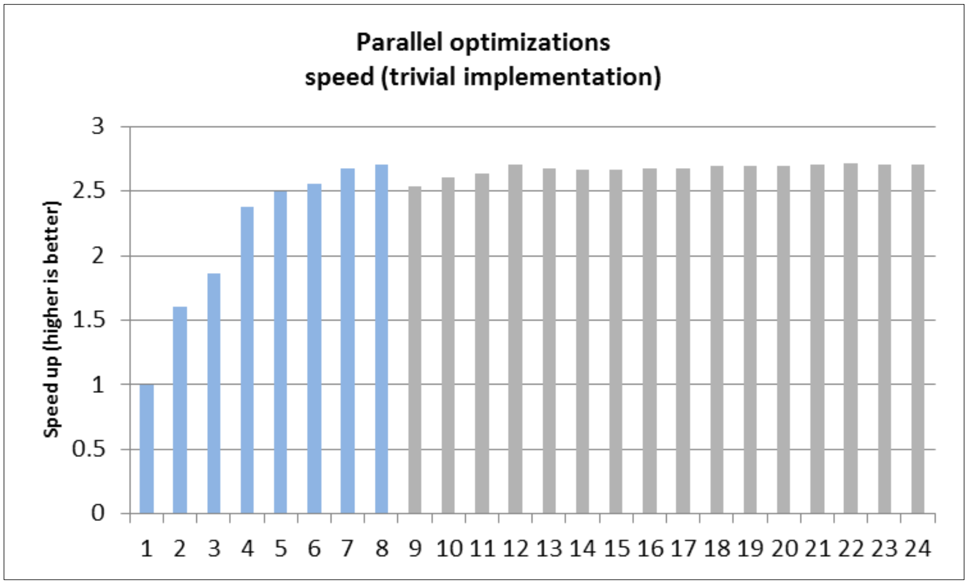


Figure 14: Parallel Speedups for the Trivial Implementation (on a 4-core HT system).

5.1.4 Instrument the Code

In this section, we present the annotation (also called instrumentation) of `popcount`. Instrumentation is important because it allows developers to analyze their application's performance, energy efficiency, etc. with little extra effort.

Do your best to limit the impact of the instrumentation on the code. We also recommend designing the instrumentation in a way that it can be easily activated/de-activated. Note that the later can be achieved at compilation time using build symbols (not done in this sample) and/or at runtime using command-line options.

5.1.4.1 Defining Counters

Define the useful work done by `popcount` as the number of checked bits. We also define a state variable that indicates when the thread(s) are productive, allowing us to assess precisely the application's energy efficiency metrics. We define the following metrics (both per thread and globally for the process):

- state
- run
- total bits checked
- total bits found set
- bits checked by each thread
- bits found set by each thread

At this stage, we know what information we want to expose via code annotation, but more importantly, we know where the data should be collected/exposed and what the scope of the data (process level or per thread level) is.

5.1.4.2 Define the Instrumentation Architecture

Following our recommendations and knowing what counters to expose, we decided to add a dedicated metrics thread to `popcount`. This thread collects the work data updated by the worker thread(s) and exposes the associated counters at regular time intervals. In our example, the most relevant user option to control the metrics thread is the update frequency. By default, our metrics thread updates the counters every second. Listing 18 shows the metrics thread's implementation code.

Lines 34 to 139

The metric thread starts by creating the data required to store the counter information.

Lines 144 to 169

Then it creates the PL with `pl_open()`.

Lines 178 to 191

These lines represent the core of the metrics thread function: the update loop. The way the loop is structured, a first counters update is always performed. This way, even if the application's run time is short and/or the update interval specified by the user is too long, a first set of data will be available to the EEP Tester.

Lines 196 to 198

When the application ends, a final update is done. This is so that the data captured by the EEP Tester will be up-to-date. Similarly, the pause at line 198 is required to allow enough time to the EEP Tester to sample the counters and to capture the final values. If your code allows such processing, we recommend that you follow these requirements.

Lines 203 to 234

Finally, the metrics thread closes the PL with `pl_close()` and ends.

```
unsigned int __stdcall metrics_thread_function(void *px) {

	//----------------------------------------------------------------------
	// Generic variables.
	//----------------------------------------------------------------------
	PMETRICS_THREAD_DATA p = NULL;
	DWORD dwret = 0;
	BOOL bret = FALSE;
	int ret = PL_FAILURE;
	size_t i = 0;

	//----------------------------------------------------------------------
	// Counters generation variables.
	//----------------------------------------------------------------------
	static size_t counter_index = 0;
	static size_t bytes_count = 0;
	static char **counters = NULL;
	char buffer[PL_MAX_PATH] = { '\0' };
	size_t memory_size = 0;

	//----------------------------------------------------------------------
	// Error variables.
	//----------------------------------------------------------------------
	char error_buffer[PL_MAX_PATH] = { '\0' };

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

	assert(px != NULL);
	p = (PMETRICS_THREAD_DATA)px;

	//----------------------------------------------------------------------
	// Allocate memory for counter data.
	//----------------------------------------------------------------------
	if(FIRST_RUN) {
		memory_size = sizeof(unsigned long long int) * p->threads_count;
		assert(memory_size > 0);
		p->per_thread_bits_checked =
			(unsigned long long int *)malloc(memory_size)
		;
		assert(p->per_thread_bits_checked != NULL);
		memset(
			p->per_thread_bits_checked,
			0,
			memory_size
		);
		p->per_thread_bits_found_set =
			(unsigned long long int *)malloc(memory_size)
		;
		assert(p->per_thread_bits_found_set != NULL);
		memset(
			p->per_thread_bits_found_set,
			0,
			memory_size
		);
		p->counters_count =
			(p->threads_count * PER_THREAD_METRICS_COUNT) +
			GLOBAL_METRICS_COUNT
		;
		memory_size = sizeof(char *) * p->counters_count;
		counters = (char **)malloc(memory_size);
		assert(counters != NULL);
		memset(
			counters,
			0,
			memory_size
		);
	}

	//----------------------------------------------------------------------
	// Set global counters.
	//----------------------------------------------------------------------
```

```
    if(FIRST_RUN) {
        counter_index = 0;
        memset(
            buffer,
            0,
            sizeof(buffer)
        );
        bytes_count = _snprintf(
            buffer,
            sizeof(buffer),
            RUN_COUNTER_NAME_STRING
        ) + 1;
        SET_COUNTER_NAME(buffer);
        memset(
            buffer,
            0,
            sizeof(buffer)
        );
        bytes_count = _snprintf(
            buffer,
            sizeof(buffer),
            BITS_CHECKED_COUNTER_NAME_STRING
        ) + 1;
        SET_COUNTER_NAME(buffer);
        memset(
            buffer,
            0,
            sizeof(buffer)
        );
        bytes_count = _snprintf(
            buffer,
            sizeof(buffer),
            BITS_FOUND_SET_COUNTER_NAME_STRING
        ) + 1;
        SET_COUNTER_NAME(buffer);
    }

    //------------------------------------------------------------------------
    // Set per-thread counters.
    //------------------------------------------------------------------------
    if(FIRST_RUN) {
        for(i = 0; i < p->threads_count; i++) {
            memset(
                buffer,
                0,
                sizeof(buffer)
            );
            bytes_count = _snprintf(
                buffer,
                sizeof(buffer),
                PER_THREAD_BITS_CHECKED_COUNTERS_NAME_FORMAT_STRING,
                i
            ) + 1;
            SET_COUNTER_NAME(buffer);
            memset(
                buffer,
                0,
                sizeof(buffer)
            );
            bytes_count = _snprintf(
                buffer,
                sizeof(buffer),
                PER_THREAD_BITS_FOUND_SET_COUNTERS_NAME_FORMAT_STRING,
                i
            ) + 1;
            SET_COUNTER_NAME(buffer);
        }
    }

    //------------------------------------------------------------------------
    // Open PL.
```

```
	//--------------------------------------------------------------------
	if(FIRST_RUN) {
		p->pld = pl_open(
			APP_NAME_STRING,
			(unsigned int)p->counters_count,
			counters,
			&p->uuid
		);
		if(p->pld == PL_INVALID_DESCRIPTOR) {
			switch(GetLastError()) {
				case PL_BYPASSED:
					break;
				case PL_FILESYSTEM_LESS_CONNECTION_FAILED:
					_snprintf(
						error_buffer,
						sizeof(error_buffer),
						ERROR_MESSAGE_FORMAT_STRING,
						ERROR_NO_PL_AGENT
					);
					printf(error_buffer);
					exit(PL_FAILURE); // for clarity only.
					break;
				default:
					assert(0);
			}
		}
	}

	//--------------------------------------------------------------------
	// Compute & expose metrics. Note that an update is forced at the end of
	// the run. This way, the latest values are exposed thru the PL. It also
	// allows to use a *very* long interval, so only 2 updates are done
	// and final values are made available. Of course, it is if you do not
	// need intermediate values to be exposed.
	//--------------------------------------------------------------------
	do {
		ret = update_metrics(p);
		assert(ret == PL_SUCCESS);
		if(h_done != NULL) {
			dwret = WaitForSingleObject(
				h_done,
				p->metrics_sampling_interval_in_ms
			);
			assert(
				(dwret == WAIT_OBJECT_0) ||
				(dwret == WAIT_TIMEOUT)
			);
		}
	} while(f_done == 0);

	//--------------------------------------------------------------------
	// Perform a last update so final values can be captured.
	//--------------------------------------------------------------------
	ret = update_metrics(p);
	assert(ret == PL_SUCCESS);
	Sleep(p->metrics_sampling_interval_in_ms * 2);

	//--------------------------------------------------------------------
	// Close PL and housekeeping.
	//--------------------------------------------------------------------
	if(LAST_RUN) {
		if(p->pld != PL_INVALID_DESCRIPTOR) {
			ret = pl_close(p->pld);
			if(ret != PL_SUCCESS) {
				switch(GetLastError()) {
					case PL_BYPASSED:
						break;
					default:
						assert(0);
				}
			}
```

```
            p->pld = PL_INVALID_DESCRIPTOR;
        }
        if(counters != NULL) {
            for(i = 0; i < p->counters_count; i++) {
                if(counters[i] != NULL) {
                    free(counters[i]);
                    counters[i] = NULL;
                }
            }
            free(counters);
            counters = NULL;
        }
        if(p->per_thread_bits_checked != NULL) {
            free(p->per_thread_bits_checked);
            p->per_thread_bits_checked = NULL;
        }
        if(p->per_thread_bits_found_set != NULL) {
            free(p->per_thread_bits_found_set);
            p->per_thread_bits_found_set = NULL;
        }
    }

    //-----------------------------------------------------------------------
    // Signal metrics thread's completion to driver.
    //-----------------------------------------------------------------------
    assert(p->h_event != NULL);
    bret = SetEvent(p->h_event);
    assert(bret == TRUE);

    return(PL_SUCCESS);
}
```

Listing 18: Implementation of the Metric Thread

The metrics thread function calls the `update_metrics()` function (shown in Listing 19) which in turn calls `pl_write()`.

The `popcount` example has only minimal error checking (essentially via assertions).

Setting the `__PL_BYPASS__` symbol causes the PL API code to compile in bypass mode. In this case, the functions will not be carried-out and will return the `PL_FAILURE` error code. The system's last error code for the calling thread is set to `PL_BYPASSED`. This is not an error, and should be distinguished from other error codes reported by the API.

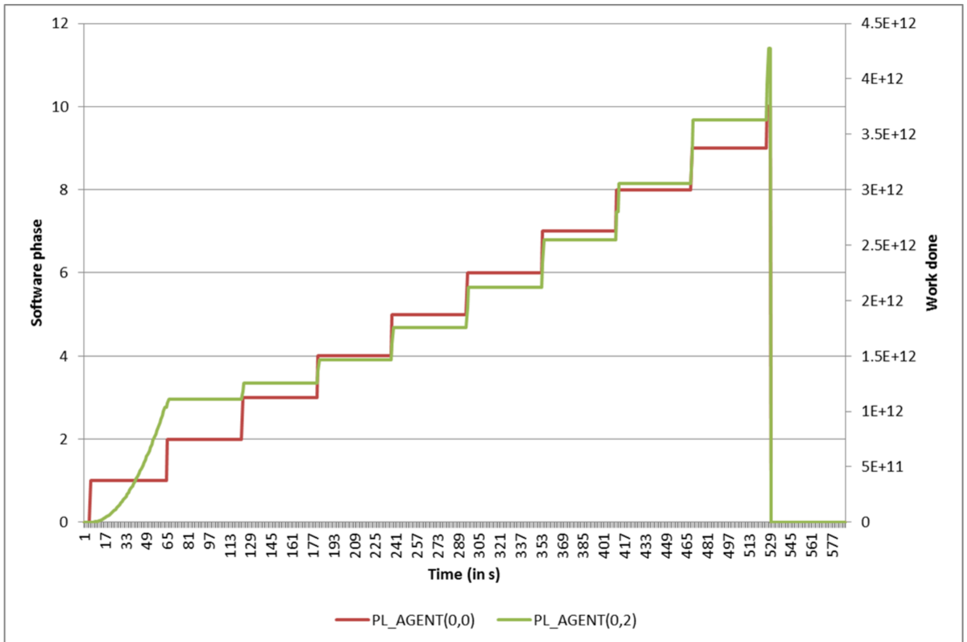
A production code should also react in a defensive way in case of API call failure (as it should with any API). Figure 15 shows an example of annotation data captured by the EEP Tester over time.

```
__forceinline PL_STATUS update_metrics(PMETRICS_THREAD_DATA p) {

    int ret = PL_FAILURE;
    size_t i = 0;
    size_t counter_index = 0;

    //-----------------------------------------------------------------------
    // Error variables.
    //-----------------------------------------------------------------------
    char error_buffer[PL_MAX_PATH] = { '\0' };

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

    assert(p != NULL);

    if(options.f_merge_runs == 0) {
        p->bits_checked = 0;
        p->bits_found_set = 0;
```

```
	}
	counter_index = BITS_FOUND_SET_COUNTER_INDEX + 1;
	for(i = 0; i < p->threads_count; i++) {
		p->bits_checked +=
			threads.p_threads_data[i].thread_counts_data.bits_checked
		;
		ret = pl_write(
			p->pld,
			&threads.p_threads_data[i].thread_counts_data.bits_checked,
			(unsigned int)counter_index
		);
		if(ret != PL_SUCCESS) {
			switch(GetLastError()) {
				case PL_BYPASSED:
					break;
				default:
					_snprintf(
						error_buffer,
						sizeof(error_buffer),
						ERROR_MESSAGE_FORMAT_STRING,
						ERROR_NO_PL_AGENT
					);
					printf(error_buffer);
					exit(PL_FAILURE); // for clarity only.
			}
		}
		counter_index++;
		p->bits_found_set +=
			threads.p_threads_data[i].thread_counts_data.bits_found_set
		;
		ret = pl_write(
			p->pld,
			&threads.p_threads_data[i].thread_counts_data.bits_found_set,
			(unsigned int)counter_index
		);
		if(ret != PL_SUCCESS) {
			switch(GetLastError()) {
				case PL_BYPASSED:
					break;
				default:
					_snprintf(
						error_buffer,
						sizeof(error_buffer),
						ERROR_MESSAGE_FORMAT_STRING,
						ERROR_NO_PL_AGENT
					);
					printf(error_buffer);
					exit(PL_FAILURE); // for clarity only.
			}
		}
		counter_index++;
	}
	ret = pl_write(
		p->pld,
		&p->bits_checked,
		BITS_CHECKED_COUNTER_INDEX
	);
	if(ret != PL_SUCCESS) {
		switch(GetLastError()) {
			case PL_BYPASSED:
				break;
			default:
				_snprintf(
					error_buffer,
					sizeof(error_buffer),
					ERROR_MESSAGE_FORMAT_STRING,
					ERROR_NO_PL_AGENT
				);
				printf(error_buffer);
				exit(PL_FAILURE); // for clarity only.
		}
```

```
	}
	ret = pl_write(
		p->pld,
		&p->bits_found_set,
		BITS_FOUND_SET_COUNTER_INDEX
	);
	if(ret != PL_SUCCESS) {
		switch(GetLastError()) {
			case PL_BYPASSED:
				break;
			default:
				_snprintf(
					error_buffer,
					sizeof(error_buffer),
					ERROR_MESSAGE_FORMAT_STRING,
					ERROR_NO_PL_AGENT
				);
				printf(error_buffer);
				exit(PL_FAILURE); // for clarity only.
		}
	}
	p->runs = threads.runs + 1;
	ret = pl_write(
		p->pld,
		&p->runs,
		RUN_COUNTER_INDEX
	);
	if(ret != PL_SUCCESS) {
		switch(GetLastError()) {
			case PL_BYPASSED:
				break;
			default:
				_snprintf(
					error_buffer,
					sizeof(error_buffer),
					ERROR_MESSAGE_FORMAT_STRING,
					ERROR_NO_PL_AGENT
				);
				printf(error_buffer);
				exit(PL_FAILURE); // for clarity only.
		}
	}

	return(PL_SUCCESS);
}
```

Listing 19: Implementation of `update_metrics()`



type	input	counter
key	PL_AGENT(0, 0)	run.
key	PL_AGENT(0, 2)	total bits found set.

Figure 15: Annotation and Work Captured by the EEP Tester

NOTE

Listing 20 shows `popcount`'s help message. For implementation details, please refer to the source code of the application.

```

 Counts the number of bits set to one in a stream of bits.

  Usage: POPCOUNT [--DELAY <d>] [--BYTES <b>] [--ITERATIONS <i>]
                  [--PARALLEL <t>] [--METRICS] [--METRICS_UPDATE <s>]
                  [--INTRINSIC | --LIBRARY | --HARDWARE]
                  [--HELP]

  --DELAY <d>:
        <d> Specifies the delay in seconds before starting processing.
        1 s by default. Must be less than 120 s.
  --BYTES <b>:
        <b> Specifies the size of the bits stream in byte(s).
        10003000 bytes by default. Must be less than 100000000 bytes.
  --ITERATIONS <i>:
        <i> Specifies the number of iterations of the processing.
        1500 iteration(s) by default. Must be less than 10000 iteration(s).
  --PARALLEL <t>:
        <t> Specifies the number of worker thread(s) used for processing.
        8 thread(s) by default. Must be less than 24 threads. if --PARALLEL
        is not specified, one worker thread is used for serial processing.
        Serial processing is the default execution mode. Using threads
        speeds-up the processing.
  --METRICS:
```

```
       Activate the metrics thread so counters are exposed. The following
       metrics are exposed:
          - run.
          - total bits checked.
          - total bits found set.
          - bits checked by thread n.
          - bits found set by thread n.
 --METRICS_UPDATE <s>:
       <s> Specifies the counters exposing interval in ms.
       1000 ms by default. Must be more than 500 ms. if --METRICS_UPDATE
       is specified, --METRICS is assumed automatically.
 --INTRINSIC | --LIBRARY | --HARDWARE:
       Use one of these options - exclusive - to specify the algorithm
       to be used during processing. --TRIVIAL is used by default.
       --INTRINSIC: uses compiler intrinsics to speed-up serial processing.
       --LIBRARY: uses optimized code (library) to speed-up serial processing.
       --HARDWARE: uses hardware acceleration to speed-up serial processing.

Examples:
       POPCOUNT
       POPCOUNT --DELAY 30
       POPCOUNT --BYTES 50000 --ITERATIONS 2000 --PARALLEL 4 --HARDWARE
       POPCOUNT --METRICS --METRICS_UPDATE 6000
```

Listing 20: `popcount` Options

At startup, `popcount` displays a summary of the options requested by the user. Once the processing is complete, a summary is displayed (Listing 21). The second section of the output is formatted as comma separated values, making it easier to copy/paste the data into a spreadsheet program for further analysis.

```

W:>popcount.exe --runs 10 --merge_runs --bytes 90000000 --iterations 1500 --parallel 4 --hardware
-----------------------------------------------
Start delay (in s):..................[1]
Problem size (in byte(s)):...........[90000000]
Iteration(s) count:..................[1500]
Parallel processing:.................[YES]
Worker thread(s) count:..............[4]
Metrics requested:...................[NO]
Metrics sampling interval (in ms):...[NA]
Processing mode:.....................[HARDWARE]
Processing function address:.........[000000013F4C4670]
-----------------------------------------------
Build mode:..........................[RELEASE]
Addressing mode:.....................[64-bit]
PL API mode:.........................[FS-LESS]
-----------------------------------------------
RUN_ID,BUILD,ADDRESSING,THREADS_COUNT,METRICS,METRICS_UPDATE_IN_MS,ALGORITHM,RUN_#,THREAD_#,START_BYTE,END_BYTE,TOTAL_
BYTES,PAYLOAD_BYTES,BITS_SCANNED,BITS_SET,TIME_IN_S
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,1,0,0,22499999,22500000,22500000,270000000000,135000000000,13.798224
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,1,3,67500000,89999999,22500000,22500000,270000000000,135000000000,13.737690
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,1,1,22500000,44999999,22500000,22500000,270000000000,135000000000,14.020513
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,1,2,45000000,67499999,22500000,22500000,270000000000,135000000000,14.290709
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,2,0,0,22499999,22500000,22500000,540000000000,270000000000,13.773930
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,2,3,67500000,89999999,22500000,22500000,540000000000,270000000000,13.838058
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,2,2,45000000,67499999,22500000,22500000,540000000000,270000000000,14.396627
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,2,1,22500000,44999999,22500000,22500000,540000000000,270000000000,14.452988
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,3,0,0,22499999,22500000,22500000,810000000000,405000000000,13.725453
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,3,1,22500000,44999999,22500000,22500000,810000000000,405000000000,13.919162
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,3,3,67500000,89999999,22500000,22500000,810000000000,405000000000,13.927385
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,3,2,45000000,67499999,22500000,22500000,810000000000,405000000000,14.127015
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,4,0,0,22499999,22500000,22500000,1080000000000,540000000000,13.716812
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-
bit,4,NO,NA,HARDWARE,4,2,45000000,67499999,22500000,22500000,1080000000000,540000000000,13.763355
```

```
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,4,3,67500000,89999999,22500000,22500000,1080000000000,540000000000,13.927472
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,4,1,22500000,44999999,22500000,22500000,1080000000000,540000000000,14.103034
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,5,1,22500000,44999999,22500000,22500000,1350000000000,675000000000,13.579931
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,5,3,67500000,89999999,22500000,22500000,1350000000000,675000000000,13.858510
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,5,2,45000000,67499999,22500000,22500000,1350000000000,675000000000,14.039397
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,5,0,0,22499999,22500000,22500000,1350000000000,675000000000,14.267947
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,6,1,22500000,44999999,22500000,22500000,1620000000000,810000000000,13.859813
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,6,0,0,22499999,22500000,22500000,1620000000000,810000000000,13.919331
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,6,3,67500000,89999999,22500000,22500000,1620000000000,810000000000,14.019222
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,6,2,45000000,67499999,22500000,22500000,1620000000000,810000000000,14.115190
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,7,1,22500000,44999999,22500000,22500000,1890000000000,945000000000,13.740260
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,7,0,0,22499999,22500000,22500000,1890000000000,945000000000,13.890040
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,7,2,45000000,67499999,22500000,22500000,1890000000000,945000000000,14.022831
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,7,3,67500000,89999999,22500000,22500000,1890000000000,945000000000,14.051338
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,8,1,22500000,44999999,22500000,22500000,2160000000000,1080000000000,13.743362
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,8,2,45000000,67499999,22500000,22500000,2160000000000,1080000000000,13.870804
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,8,0,0,22499999,22500000,22500000,2160000000000,1080000000000,14.023790
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,8,3,67500000,89999999,22500000,22500000,2160000000000,1080000000000,14.112286
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,9,1,22500000,44999999,22500000,22500000,2430000000000,1215000000000,13.833373
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,9,3,67500000,89999999,22500000,22500000,2430000000000,1215000000000,13.838359
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,9,0,0,22499999,22500000,22500000,2430000000000,1215000000000,14.120309
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,9,2,45000000,67499999,22500000,22500000,2430000000000,1215000000000,14.201006
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,10,1,22500000,44999999,22500000,22500000,2700000000000,1350000000000,13.824216
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,10,0,0,22499999,22500000,22500000,2700000000000,1350000000000,13.865574
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,10,2,45000000,67499999,22500000,22500000,2700000000000,1350000000000,14.128411
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,10,3,67500000,89999999,22500000,22500000,2700000000000,1350000000000,14.273878
98dc2eca-6e4e-4e64-8f92-521433a8e498,RELEASE,64-bit,4,NO,NA,HARDWARE,*,*,*,*,*,*,10800000000000,5400000000000,142.161770
```

Listing 21: Typical `popcount` Output

5.1.4.3 Measuring EEP with Annotation

For our reference run, we use `popcount` options shown in Listing 22. For our comparison run, we use `popcount` options shown in Listing 23. Figure 16 shows the relative EEP improvement between the reference and comparison runs on our test system. Improving the EEP by ~16X can be considered as a successful EEP optimization project.

```
--bytes 90000000 --iterations 100 --parallel 1 --runs 10 --merge_runs –metrics
```

Listing 22: Options for Reference Run

```
--bytes 90000000 --iterations 100 --parallel 4 --runs 10 --merge_runs –metrics –hardware
```

Listing 23: Options for Validation Run

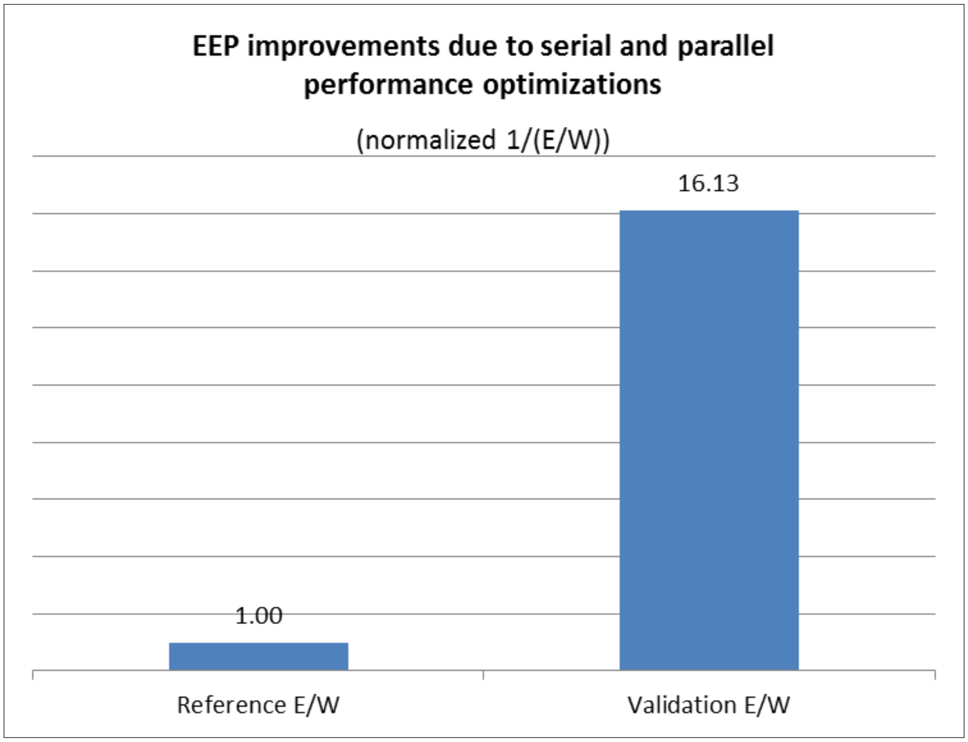


Figure 16: Relative EEP Improvement of `popcount`.

6 Using the EEP Tester

This section describes how to use the EEP Tester to analyze your application.

The examples shown in this section use the Microsoft* Windows built-in batch interpreter. It can be adapted to any other scripting facilities you are using.

The overall processing flow is:

1. Recharge your batteries (80% minimum) and disconnect the power transformer.
2. Start the EEP Tester.
3. Start the instrumented workload.
4. Share the application's `pid` with the EEP Tester.
5. Exercise your application (in a deterministic way).
6. At the end of the workload run, stop the EEP Tester.
7. Save and analyze the data.

This section details the operations listed in steps 2 to 6 as they are implemented in a set of demonstration batch files listed below. Studying these scripts shows the basics of how you can drive the EEP Tester from an automation framework.

- `go.bat`
- `optimized_instrumented.bat`
- `start_esrv.bat`
- `unoptimized_instrumented.bat`

6.1.1 The `go.bat` File

This script is the driver. It performs two iterations of the operations listed earlier (steps 2 to 6) for the same workload in an unoptimized and optimized way (using the `popcount` sample application). Refer to Listing 24.

Lines 5 to 11 (shown in dark blue)

Set key variables used to easily configure the scripts. For example, the `BIN_DIR` variable is used to point to the location where the EEP Tester binaries are installed. The values used in these scripts should be usable as-is.

Line 13 (shown in red)

Clears the `OUTPUT_DIR` folder. This is important because subsequent batch files make the assumption that this folder is empty.

Line 16 (shown in purple)

Runs `start_esrv.bat` which directs that the workload run in an unoptimized way.

Lines 17 to 18 (shown in purple)

Renames output files. Note the `/WAIT` option used with `start` on line 16. This halts script execution until `start_esrv.bat` completes.

Line 20

Pauses for 5 seconds. This pause is important because power and energy have a measurable inertia. By inertia is meant that once your workload starts and taxes the processor, the power draw increase does not happen until a few seconds later.

Similarly, when the workload ends, the power draw decays to the idle level in few seconds.

The pause time may require tweaking on different systems. Pausing helps ensure that the data collection runs under similar conditions for the unoptimized and the optimized versions of the workload.

Lines 23 to 25

The optimized processing of the workload is performed.

```
1   rem run all tests in sequence
2   @echo off
3   cls
4
5   set DEBUG=yes
6   set BIN_DIR=C:\EEPC\run\64
7   set CONFIGS_DIR=C:\EEPC\configuration_files
8   set OUTPUT_DIR=C:\EEPC\run\outputs
9   set ESRV_ADDRESS=127.0.0.1
10  set ESRV_PORT=49260
11  set WORKLOAD=popcount.exe
12
13  del /F /Q %OUTPUT_DIR%
14
15  echo Running UNOPTIMIZED, INSTRUMENTED Version.
16  start /WAIT "Running UNOPTIMIZED, INSTRUMENTED Version" start_esrv.bat
    unoptimized_instrumented.bat %WORKLOAD%
17  move /Y %OUTPUT_DIR%\test_key-000000.csv %OUTPUT_DIR%\UNOPTIMIZED_INSTRUMENTED_key.csv >
    nul
18  move /Y %OUTPUT_DIR%\test-000000.csv %OUTPUT_DIR%\UNOPTIMIZED_INSTRUMENTED.csv > nul
19
20  timeout /t 5
21
22  echo Running OPTIMIZED, INSTRUMENTED Version.
23  start /WAIT "Running OPTIMIZED, INSTRUMENTED Version" start_esrv.bat
    optimized_instrumented.bat %WORKLOAD%
24  move /Y %OUTPUT_DIR%\test_key-000000.csv %OUTPUT_DIR%\OPTIMIZED_INSTRUMENTED_key.csv >
    nul
25  move /Y %OUTPUT_DIR%\test-000000.csv %OUTPUT_DIR%\OPTIMIZED_INSTRUMENTED.csv > nul
26
27  echo Done! You can now analyze data in [%OUTPUT_DIR%] folder.
28
29  @echo on
```

Listing 24: `go.bat`

6.1.2 The `optimized_instrumented.bat` File

After the variables setting (lines 5 to 11), this script starts the workload at lines 13-14 (step 5). This is where you could call your own workload automation script or program.

Lines 16 to 21 are used to perform step 6 of our tasks list. Refer to Listing 25.

```
1   rem run a workload, then stops ESRV
2   @echo off
3   cls
4
5   set DEBUG=yes
6   set BIN_DIR=C:\EEPC\run\64
7   set CONFIGS_DIR=C:\EEPC\configuration_files
8   set WORKLOAD_BIN_DIR=C:\EEPC\MSVCsolutions\MSVC2010\popcount\x64\Release
9   set WORKLOAD=popcount.exe
10  set ESRV_ADDRESS=127.0.0.1
11  set ESRV_PORT=49260
12
```

```
%WORKLOAD_BIN_DIR%\%WORKLOAD% --delay 10 --metrics --hardware --parallel 2 --bytes
9000000 --iterations 2000 --runs 1

set COMMAND="%BIN_DIR%\esrv"^
--stop^
--address %ESRV_ADDRESS%^
--port %ESRV_PORT%
if %DEBUG% == YES echo COMMAND=[%COMMAND%]
%COMMAND%

exit
@echo on
```

Listing 25: `optimized_instrumented.bat`

6.1.3 The `unoptimized_instrumented.bat` File

After the variables setting (lines 5 to 11), this script starts the workload at line 13 (step 5). This is where you could call your own workload automation script or program.

Lines 15 to 20 are used to perform step 6 of our tasks list. Refer to Listing 26.

```
1   rem run a workload, then stops ESRV
2   @echo off
3   cls
4
5   set DEBUG=yes
6   set BIN_DIR=C:\EEPC\run\64
7   set CONFIGS_DIR=C:\EEPC\configuration_files
8   set WORKLOAD_BIN_DIR=C:\EEPC\MSVCsolutions\MSVC\popcount\x64\Release
9   set WORKLOAD=popcount.exe
10  set ESRV_ADDRESS=127.0.0.1
11  set ESRV_PORT=49260
12
13  %WORKLOAD_BIN_DIR%\%WORKLOAD% --delay 10 --metrics --parallel 1 --bytes 9000000 --
    iterations 2000 --runs 1
14
15  set COMMAND="%BIN_DIR%\esrv"^
16  --stop^
17  --address %ESRV_ADDRESS%^
18  --port %ESRV_PORT%
19  if %DEBUG% == YES echo COMMAND=[%COMMAND%]
20  %COMMAND%
21
22  exit
23  @echo on
```

Listing 26: `unoptimized_instrumented.bat`

6.1.4 The `start_esrv.bat` File

`start_esrv.bat`, shown in Listing 27, is the most complex and interesting script of these examples. After the variable setting (lines 5 to 11), this script starts the EEP Tester (lines 12 to 32). At line 39, the workload is started. %1 is the first argument of this script, since this script is called from lines 16 and 23 in the `go.bat` script. Therefore, %1 is the name of the script that launched the workload (`unoptimized_instrumented.bat`, for example).

The next key code block is listed at lines 41-47. The %2 is the name of the process image name, which, in our case, is `popcount.exe`. This name is used to detect if the application has started. We recommend that the application not start processing immediately This is even more important when concurrency data are required. Remember that when collecting concurrency data, the EEP Tester needs to know the process ID (the `pid`). The pause of 5 seconds at line 49 allows the EEP Tester to start-up and initialize. If this pause is not taken, the EEP Tester may miss the `pid` when sent from lines 51 to 57.

Finally, lines 59 to 66 just checks for the end of the application, using the same method (note the “not” in line 66). The granularity of these two loops is set to 10 seconds, and this value can be adapted. Ten seconds is long enough not to interfere with the measurement (although a less verbose waiting program could be used), and is short enough not to have to wait too long before detecting the application’s end. However, it is important to understand that when an instrumented application that exposes a state variable and indicates precisely when it starts to run and stops, it completely removes the impact of this parameter on the measurements. Instrumenting your application is strongly recommended..

```
1   rem start workload.bat executable_name
2   @echo off
3   cls
4
5   set DEBUG=YES
6   set BIN_DIR=C:\EEPC\run\64
7   set CONFIGS_DIR=C:\EEPC\configuration_files
8   set OUTPUT_DIR=C:\EEPC\run\outputs
9   set ESRV_ADDRESS=127.0.0.1
10  set ESRV_PORT=49260
11
12  set COMMAND="%BIN_DIR%\esrv.exe"^
13   --start^
14   %DIAGNOSTIC%^
15   --time_in_ms^
16   --pause 1000^
17   --library "%BIN_DIR%\intel_modeler.dll"^
18   --no_pl^
19   --kernel_priority_boost^
20   --no_auto_port^
21   --device_options^
22   ^"^
23   mode=1^
24   time=no^
25   output=test^
26   output_format=csv^
27   format_output=yes^
28   process=yes^
29   output_folder='%OUTPUT_DIR%'^
30   output_buffer=1024^
31   power='%CONFIGS_DIR%\power\power_config.txt'^
32   os='%CONFIGS_DIR%\os\os_counters.txt'^
33   pl_agent='%CONFIGS_DIR%\pl_agent\pl_agent.txt'^
34   il='%BIN_DIR%\concurrency_input.dll'^
35   "
36  echo COMMAND = [%COMMAND%]
37  start "ESRV" %COMMAND%
38
39  start "WORKLOAD" %1
40
41  set COMMAND=tasklist /FI "IMAGENAME eq %2"
42  if %DEBUG% == YES echo COMMAND=[%COMMAND%]
43  :redo
44  for /F "tokens=2" %%A in ('%COMMAND%') do (
45        set APP_PID=%%A
46  )
47  if %APP_PID% == No goto redo
48
49  timeout /t 5
50
51  set COMMAND="%BIN_DIR%\esrv.exe"^
52   --device_control PIDS %APP_PID%^
53   --address %ESRV_ADDRESS%^
54   --port %ESRV_PORT%^
```

```
 %DIAGNOSTIC%
if %DEBUG% == YES echo COMMAND=[%COMMAND%]
%COMMAND%

set COMMAND=tasklist /FI "IMAGENAME eq %2"
if %DEBUG% == YES echo COMMAND=[%COMMAND%]
:wait
timeout /t 10
for /F "tokens=2" %%A in ('%COMMAND%') do (
        set APP_PID=%%A
)
if not %APP_PID% == No goto wait

exit
@echo on
```

Listing 27: `start_esrv.bat`

Figure 17 shows the power draw and CPU utilization as extracted from the data collected with the demo scripts just described.

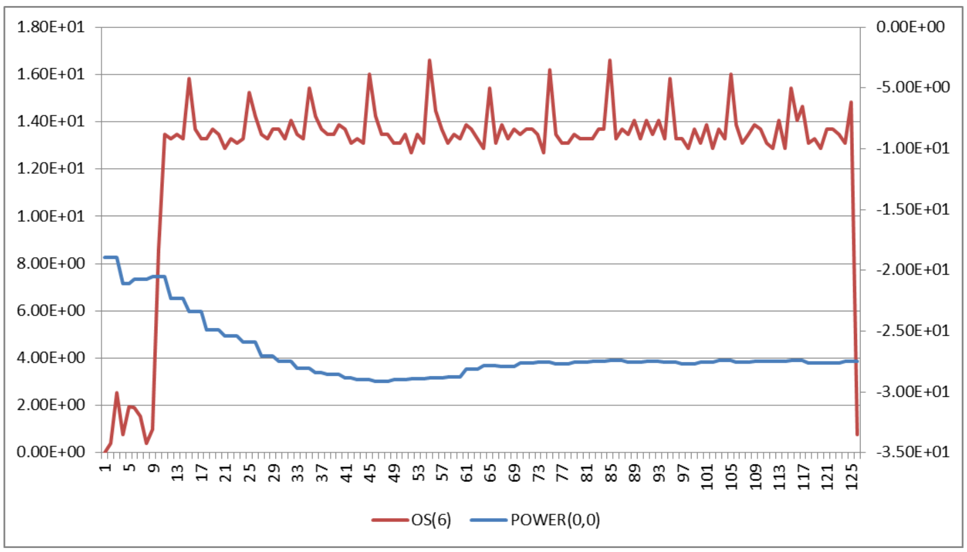


Figure 17: Power Draw and CPU Utilization vs. Time

7 Appendix

7.1 PL Protocol

The PL protocol is a simple network protocol designed to encapsulate and send API calls to a networked agent and to receive and decode a networked agent's answer to the API calls. The EEP Tester is using its own embedded PL Agent. There is no need to run an agent on the test system to use the API in file system-less mode. From an application point of view, there are no functional differences between the file-system-based and the file system-less mode.

If the agent and the instrumented applications are not running on the same node, there may be network overhead and jitter to be considered when using the API in file system-less mode. The application should also handle error conditions thoroughly since a network is generally less reliable than a file system.

NOTE

> Information provided on the protocol itself is useful only when implementing an agent, or building a server into an application. Many details provided in this section target server developers willing to add support for the PL protocol to their software and serve API calls autonomously.

7.2 PL Message Format

A PL message is a well-defined string of bytes. Non-textual data are encoded as binary data. Binary data are encoded in LE (little-endian) order. This means that the LSB (Least Significant Byte) is the first byte, and the MSB (Most Significant Byte) is the last byte received. A well-formed PL message is composed of a:

- Header (4 bytes)
- Body (variable size, in bytes)
- EOR – End of Record (1 byte)

NOTE

> The message header encodes the size (in bytes) of the body and the end of record. This size doesn't include the size of the header itself. For example, a header, a message body and the end of record composed of 100 bytes will have its first four bytes equal to 0x60 0x00 0x00 0x00 (96 in decimal). This allows for a fast send/receive mechanism. Indeed, it suffices for the receiver to read four bytes from a socket to know how many bytes must be read to fully receive a message. The emitter sends the entire message in a single operation.

7.2.1 Status Code

The protocol defines a `PL_PROTOCOL_STATUS`. A PL protocol status can be equal to `PL_PROTOCOL_FAILURE` or to `PL_PROTOCOL_SUCCESS`. These codes are used by the agent and the API code compiled in file system-less mode. Applications using the core API do not need to refer to these status codes. Instead, they must use `PL_STATUS` (`PL_FAILURE` or `PL_SUCCESS`). If a network-related or protocol-related error is detected, then `PL_FAILURE` is returned by the function, and the system error code is set accordingly.

7.2.1.1 String Encoding

The API uses strings to represent the application name, the counter names, and the PL configuration file name and path. A well-formed string is composed of the following.

- Header (4 bytes, encoding the value N, number of characters)
- Body (N bytes, each of them encoding a single character)

NOTE

Strings are NOT terminated by a null character.

NOTE

In this section, the following color codes are used to highlight message elements:

Header
Operation code
UUID
PL descriptor
String size
String data
Counter count
Counter offset
Counter value
Status
End of record

7.2.2 `pl_open()` Encoding

When called in file system-less mode, `pl_open()` builds a PL message with a body composed of the following.

- Operation code (1 byte)
- Counter count (4 bytes, encoding the value N – number of counters)
- String (variable size – application name)
- N Strings (variable size – counter names)

An agent should return a message with a body composed of the following:

- Status code (4 bytes)
- `UUID` (16 bytes)
- PL descriptor (4 bytes)

As an example, assume the following call to `pl_open()`.Note that only relevant data are shown in Listing 28.

```
const char *counters[5] = {
   "Hello",
   "World",
   NULL,
   "A",
   "b"
};

pld = pl_open(
   "Application in filesystem-less mode",
```

```
    5,
    counters,
    &uuid
);
```

Listing 28: Call to `pl_open()`

Listing 29 is a HEX memory dump that shows the PL message sent to the agent. Addresses are arbitrary.

```
0x0012DC38  60 00 00 00 01 05 00 00 00 23 00 00 00 41 70 70  `........#...App
0x0012DC48  6c 69 63 61 74 69 6f 6e 20 69 6e 20 66 69 6c 65  lication in file
0x0012DC58  73 79 73 74 65 6d 2d 6c 65 73 73 20 6d 6f 64 65  system-less mode
0x0012DC68  05 00 00 00 48 65 6c 6c 6f 05 00 00 00 57 6f 72  ....Hello....Wor
0x0012DC78  6c 64 13 00 00 00 61 6e 6f 6e 79 6d 6f 75 73 5f  ld....anonymous_
0x0012DC88  63 6f 75 6e 74 65 72 5f 32 01 00 00 00 41 01 00  counter_2....A..
0x0012DC98  00 00 62 0d 00 00 00 00 00 00 00 00 00 00 00 00  ..b.............
```

Listing 29: HEX Memory Dump of PL Message Sent to Agent (`pl_open()`)

The returned PL descriptor is zero (0) and the `UUID` is `CF8C9562-561D-4A20-A5BB-443061652328`. The call is successful (also visible in the PL protocol status as `PL_PROTOCOL_SUCCESS`). Listing 30 is a HEX memory dump that shows the PL message received from the agent. Addresses are arbitrary.

```
0x0012EC40  19 00 00 00 00 00 00 00 62 95 8c cf 1d 56 20 4a  ........b.ŒÏ.V J
0x0012EC50  a5 bb 44 30 61 65 23 28 01 00 00 00 0d 00 00 00  ¥»D0ae#(........
```

Listing 30: HEX Memory Dump of PL Message received from Agent (`pl_open()`)

When compiled in debug mode, the sample agent prints a HEX dump and a clear decode of the PL messages received from clients and sent to the clients as well as other data. This is a facility that can be used to analyze the use of the PL protocol. Listing 31 shows the log extract for the previous example.

```
Pool thread [0] is serving a PL API call.
 ...Pool thread [0] has received...
 ......Pool thread [0]: Bytes in full message: [100]d - [64]h.
 ......Pool thread [0]: Bytes in message (skipping size header): [96]d - [60]h.
 ......Pool thread [0]: 60 00 00 00 01 05 00 00 00 23 00 00 00 41 70 70 6C 69 63
 61 74 69 6F 6E 20 69 6E 20 66 69 6C 65 73 79 73 74 65 6D 2D 6C 65 73 73 20 6D
 6F 64 65 05 00 00 00 48 65 6C 6C 6F 05 00 00 00 57 6F 72 6C 64 13 00 00 00 61
 6E 6F 6E 79 6D 6F 75 73 5F 63 6F 75 6E 74 65 72 5F 32 01 00 00 00 41 01 00 00
 00 62 0D
 ......Pool thread [0]: xx xx xx xx 01 05 00 00 00 23 00 00 00 41 70 70 6C 69 63
 61 74 69 6F 6E 20 69 6E 20 66 69 6C 65 73 79 73 74 65 6D 2D 6C 65 73 73 20 6D
 6F 64 65 05 00 00 00 48 65 6C 6C 6F 05 00 00 00 57 6F 72 6C 64 13 00 00 00 61
 6E 6F 6E 79 6D 6F 75 73 5F 63 6F 75 6E 74 65 72 5F 32 01 00 00 00 41 01 00 00
 00 62 0D
 ......Pool thread [0]: Op code = [PL_PROTOCOL_OPCODE_OPEN].
 ......Pool thread [0]: Counters count = [5].
 ......Pool thread [0]: Application name length = [35].
 ......Pool thread [0]: Application name = [Application in filesystem-less
 mode].
 ......Pool thread [0]: Counter [0] length = [5].
 ......Pool thread [0]: Counter [0] name = [Hello].
 ......Pool thread [0]: Counter [1] length = [5].
 ......Pool thread [0]: Counter [1] name = [World].
 ......Pool thread [0]: Counter [2] length = [19].
 ......Pool thread [0]: Counter [2] name = [anonymous_counter_2].
```

```
......Pool thread [0]: Counter [3] length = [1].
......Pool thread [0]: Counter [3] name = [A].
......Pool thread [0]: Counter [4] length = [1].
......Pool thread [0]: Counter [4] name = [b].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is sending...
......Pool thread [0]: Bytes in full message: [29]d - [1d]h.
......Pool thread [0]: Bytes in message (skipping size header): [25]d - [19]h.
......Pool thread [0]: 19 00 00 00 00 00 00 00 62 95 8C CF 1D 56 20 4A A5 BB 44 30 61 65 23 28 01 00 00 00 0D
......Pool thread [0]: xx xx xx xx 00 00 00 00 62 95 8C CF 1D 56 20 4A A5 BB 44 30 61 65 23 28 01 00 00 00 0D
......Pool thread [0]: Status = [PL_PROTOCOL_SUCCESS].
......Pool thread [0]: Answer to op code = [PL_PROTOCOL_OPCODE_OPEN].
......Pool thread [0]: uuid = [cf8c9562-561d-4a20-a5bb-443061652328].
......Pool thread [0]: pld = [1].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
```

Listing 31: Log Extract (`pl_open()`)

In file system-less mode, two important items need to be remembered.

First, in file system-less mode, the client (the application) and the server (the agent) have different PL descriptors. In the example, the client PL descriptor is 0 and the server descriptor is 1. This is due to the fact that the agent is serving multiple clients and that its PL descriptor table may already be in use. In this example, another client already performed a call to `pl_open()` in file system-less mode. However, the `pl_open()` call is the first for this client and, predictably, the PL descriptor return is 0.

Second, the `UUID` is different between the client and the server. The client `UUID` is `CF8C9562-561D-4A20-A5BB-443061652328` and the server `UUID` is `CF8C9562-561D-4A20-A5BB-443061652328`. The cause of this discrepancy is the same as for the PL descriptor discrepancy. The API is automatically performing the mapping between the client and the server PL descriptors and `UUID`s.

7.2.3 `pl_close()` Encoding

When called in file system-less mode, `pl_close()` builds a PL message with a body composed of the following.

- Operation code (1 byte)
- `UUID` (16 bytes)
- PL descriptor (4 bytes)

An agent should return a message with a body composed of the following.

- Status Code (4 bytes)

Assume the following call to `pl_close()`. Note that only relevant data are shown in Listing 32.

```
pld = pl_open(
    "Application in filesystem-less mode",
    5,
    counters,
    &uuid
);
pl_ret = pl_close(pld);
```

Listing 32: Call to `pl_close()`

Listing 33 is a HEX memory dump that shows the PL message sent to the agent. Addresses are arbitrary.

```
0x0012DD64  16 00 00 00 03 81 ca 56 92 a3 97 f7 4b 8b 37 f1   ......ÊV’£—÷K.7ñ
0x0012DD74  a0 51 68 8e c8 00 00 00 00 0d 00 00 00 00 00 00   QhŽÈ..........
```

Listing 33: HEX Memory Dump of PL Message Sent to Agent (`pl_close()`)

The returned status is `PL_SUCCESS`. The call is successful (also visible in the PL protocol status as `PL_PROTOCOL_SUCCESS`). Listing 34 is a HEX memory dump that shows the PL message received from the agent. Addresses are arbitrary.

```
0x0012ED6C  05 00 00 00 00 00 00 00 0d 00 00 00 00 00 00 00   ................
```

Listing 34: HEX Memory Dump of PL Message received from Agent (`pl_close()`)

When compiled in debug mode, the sample agent prints a HEX dump of the messages received and the messages sent as well as other data. This is a facility that can be used to analyze the use of the PL protocol. Listing 35 shows the log for the previous example.

```
.Pool thread [0] has received...
 ....Pool thread [0]: Bytes in full message: [26]d - [1a]h.
 ....Pool thread [0]: Bytes in message (skipping size header): [22]d - [16]h.
 ....Pool thread [0]: 16 00 00 00 03 81 CA 56 92 A3 97 F7 4B 8B 37 F1 A0 51 68
8E C8
 00 00 00 00 0D
 ....Pool thread [0]: xx xx xx xx 03 81 CA 56 92 A3 97 F7 4B 8B 37 F1 A0 51 68
8E C8
 00 00 00 00 0D
 ....Pool thread [0]: Op code = [PL_PROTOCOL_OPCODE_CLOSE].
 ....Pool thread [0]: uuid = [761f44a8-2409-4d9e-bb56-2234718c03a1].
 ....Pool thread [0]: pld = [0].
 ....Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
 .Pool thread [0] is sending...
 ....Pool thread [0]: Bytes in full message: [9]d - [9]h.
 ....Pool thread [0]: Bytes in message (skipping size header): [5]d - [5]h.
 ....Pool thread [0]: 05 00 00 00 00 00 00 00 0D
 ....Pool thread [0]: xx xx xx xx 00 00 00 00 0D
 ....Pool thread [0]: Status = [PL_PROTOCOL_SUCCESS].
 ....Pool thread [0]: Answer to op code = [PL_PROTOCOL_OPCODE_CLOSE].
 ....Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
```

Listing 35: Log Extract (`pl_close()`)

7.2.4 `pl_write()` Encoding

When called in file system-less mode, `pl_write()` builds a PL message with a body composed of the following.

- Operation code (1 byte)

- `UUID` (16 bytes)
- PL descriptor (4 bytes)
- Counter offset (4 bytes)
- Counter value (8 bytes)

An agent should return a message with a body composed of the following.

- Status Code (4 bytes)

Assume the following call to `pl_write()`. Note that only relevant data are shown in Listing 36.

```
pld = pl_open(
   "Application in filesystem-less mode",
   5,
   counters,
   &uuid
);

value = PL_MAX_COUNTER_VALUE;

pl_ret = pl_write(
   pld,
   &value,
   1
);

pl_ret = pl_close(pld);
```

Listing 36: Call to `pl_write()`

Listing 37 is a HEX memory dump shows the PL message sent to the agent. Addresses are arbitrary.

```
0x0012DD58  22 00 00 00 05 26 0a 09 90 3e f0 cf 48 8d b3 d2  "....&...>ðÏH..Ò
0x0012DD68  2e 11 3c 38 d1 00 00 00 00 01 00 00 00 ff ff ff  ..<8Ñ........ÿÿÿ
0x0012DD78  ff ff ff ff ff 0d 00 00 00 00 00 00 00 00 00 00  ÿÿÿÿÿ...........
```

Listing 37: HEX Memory Dump of PL Message Sent to Agent (`pl_write()`)

The returned status is `PL_SUCCESS`. The call is successful (also visible in the PL protocol status as `PL_PROTOCOL_SUCCESS`). Listing 38 is a HEX memory dump that shows the PL message received from the agent. Addresses are arbitrary.

```
0x0012ED60  05 00 00 00 00 00 00 00 0d 00 00 00 00 00 00 00  ................
```

Listing 38: HEX Memory Dump of PL Message received from Agent (`pl_write()`)

When compiled in debug mode, the sample agent prints a HEX dump of the messages received and the messages sent as well as other data. This is a facility that can be used to analyze the use of the PL protocol. Listing 39 shows the log for the previous example.

```
.Pool thread [0] has received...
....Pool thread [0]: Bytes in full message: [38]d - [26]h.
....Pool thread [0]: Bytes in message (skipping size header): [34]d - [22]h.
....Pool thread [0]: 22 00 00 00 05 26 0A 09 90 3E F0 CF 48 8D B3 D2 2E 11 3C
38 D1
 00 00 00 00 01 00 00 00 FF FF FF FF FF FF FF FF 0D
....Pool thread [0]: xx xx xx xx 05 26 0A 09 90 3E F0 CF 48 8D B3 D2 2E 11 3C
38 D1
 00 00 00 00 01 00 00 00 FF FF FF FF FF FF FF FF 0D
```

```
....Pool thread [0]: Op code = [PL_PROTOCOL_OPCODE_WRITE].
....Pool thread [0]: uuid = [90090a26-f03e-48cf-8db3-d22e113c38d1].
....Pool thread [0]: pld = [0].
....Pool thread [0]: counter offset = [1].
....Pool thread [0]: counter value = [18446744073709551615].
....Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
.Pool thread [0] is sending...
....Pool thread [0]: Bytes in full message: [9]d - [9]h.
....Pool thread [0]: Bytes in message (skipping size header): [5]d - [5]h.
....Pool thread [0]: 05 00 00 00 00 00 00 00 0D
....Pool thread [0]: xx xx xx xx 00 00 00 00 0D
....Pool thread [0]: Status = [PL_PROTOCOL_SUCCESS].
....Pool thread [0]: Answer to op code = [PL_PROTOCOL_OPCODE_WRITE].
....Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
```

Listing 39: Log Extract (`pl_write()`)

7.2.5 Complete Transaction Example

Assume the program shown in Listing 40. Note that only relevant data are shown in the listing below.

```
Unsigned long long int value = PL_MAX_COUNTER_VALUE;
const char *counters[5] = {
   "Hello",
   "World",
   NULL,
   "A",
   "b"
};

pld = pl_open(
   "Application in filesystem-less mode",
   5,
   counters,
   &uuid
);

pl_ret = pl_write(
   pld,
   &value,
   1
);

value = PL_MIN_COUNTER_VALUE;

pl_ret = pl_close(pld);
```

Listing 40: Complete Transaction Example

When compiled in debug mode, the sample agent prints a HEX dump of the messages received and the messages sent as well as other data. This is a facility that can be used to analyze the use of the PL protocol. Listing 41 shows the log for the previous, full transaction example. Note that to limit the log size, the sample agent was configured with a single worker thread in the pool.

```
pl_agent has started.
Parsing user input.
pl_agent version [2010.06.08].
Using PL helper version [2009.05.18].
Using PL version [2010.12.15(W)].
Initializing Windows socket system.
Agent is running on [10.24.0.35].
ADMIN port is [49252] and PL port is [49253].
Allocating thread pool data.
Creating synchronization objects.
Creating thread pool.
```

```
Agent has [1] thread(s) in the pool.
Creating admin port listener thread.
Creating pl port listener thread.
Pool thread [0] has started.
To interrupt the agent, type <CTRL>+<C>.
Pool thread [0] is waiting for main thread to be done.
Signaling main thread done.
Waiting for all threads to end.
Pool thread [0] has received the main thread done signal.
Pool thread [0] is waiting for a PL API call to serve.
Admin port listener thread has started.
Pl port listener thread has started.
Admin port listener thread is waiting for main thread to be done.
Pl port listener thread is waiting for main thread to be done.
Admin port listener thread has received the main thread done signal.
Pl port listener thread has received the main thread done signal.
Admin port listener thread is setting-up IPC.
Pl port listener thread is setting-up IPC.
...Admin port listener thread is initializing IPC data.
...Pl port listener thread is initializing IPC data.
...Admin port listener thread is setting-up socket IPC data.
...Pl port listener thread is setting-up socket IPC data.
...Admin port listener thread is resolving IPC address & port.
...Pl port listener thread is resolving IPC address & port.
...Admin port listener thread is attempting to create & bind IPC socket.
...Pl port listener thread is attempting to create & bind IPC socket.
...Pl port listener thread is listening on IPC bound socket.
...Admin port listener thread is listening on IPC bound socket.
...Admin port listener thread is accepting connections.
...Pl port listener thread is accepting connections.
...Pl port listener thread has received a request.
...Pl port listener thread is searching a thread in the pool to serve the request.
......Pl port listener thread is trying to lock pool thread [0].
......Pl port listener thread has successfully locked pool thread [0].
...Pl port listener thread has triggered pool thread [0].
...Pl port listener thread is accepting connections.
Pool thread [0] is serving a PL API call.
.Pool thread [0] has received...
....Pool thread [0]: Bytes in full message: [100]d - [64]h.
....Pool thread [0]: Bytes in message (skipping size header): [96]d - [60]h.
....Pool thread [0]: 60 00 00 00 01 05 00 00 00 23 00 00 00 41 70 70 6C 69 63 61 74
69 6F 6E 20 69 6E 20 66 69 6C 65 73 79 73 74 65 6D 2D 6C 65 73 73 20 6D 6F 64 65 05 00
00 00 48 65 6C 6C 6F 05 00 00 00 57 6F 72 6C 64 13 00 00 00 61 6E 6F 6E 79 6D 6F 75 73
5F 63 6F 75 6E 74 65 72 5F 32 01 00 00 00 41 01 00 00 00 62 0D
......Pool thread [0]: xx xx xx xx 01 05 00 00 00 23 00 00 00 41 70 70 6C 69 63 61 74
69 6F 6E 20 69 6E 20 66 69 6C 65 73 79 73 74 65 6D 2D 6C 65 73 73 20 6D 6F 64 65 05 00
00 00 48 65 6C 6C 6F 05 00 00 00 57 6F 72 6C 64 13 00 00 00 61 6E 6F 6E 79 6D 6F 75 73
5F 63 6F 75 6E 74 65 72 5F 32 01 00 00 00 41 01 00 00 00 62 0D
......Pool thread [0]: Op code = [PL_PROTOCOL_OPCODE_OPEN].
......Pool thread [0]: Counters count = [5].
......Pool thread [0]: Application name length = [35].
......Pool thread [0]: Application name = [Application in filesystem-less mode].
......Pool thread [0]: Counter [0] length = [5].
......Pool thread [0]: Counter [0] name = [Hello].
......Pool thread [0]: Counter [1] length = [5].
......Pool thread [0]: Counter [1] name = [World].
......Pool thread [0]: Counter [2] length = [19].
......Pool thread [0]: Counter [2] name = [anonymous_counter_2].
......Pool thread [0]: Counter [3] length = [1].
......Pool thread [0]: Counter [3] name = [A].
......Pool thread [0]: Counter [4] length = [1].
......Pool thread [0]: Counter [4] name = [b].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is sending...
......Pool thread [0]: Bytes in full message: [29]d - [1d]h.
......Pool thread [0]: Bytes in message (skipping size header): [25]d - [19]h.
......Pool thread [0]: 19 00 00 00 00 00 00 00 90 4D A5 D1 26 A5 BD 40 85 47 6A F8 CD
07 DA 18 00 00 00 00 0D
......Pool thread [0]: xx xx xx xx 00 00 00 00 90 4D A5 D1 26 A5 BD 40 85 47 6A F8 CD
07 DA 18 00 00 00 00 0D
......Pool thread [0]: Status = [PL_PROTOCOL_SUCCESS].
......Pool thread [0]: Answer to op code = [PL_PROTOCOL_OPCODE_OPEN].
......Pool thread [0]: uuid = [d1a54d90-a526-40bd-8547-6af8cd07da18].
......Pool thread [0]: pld = [0].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is closing IPC.
...Pool thread [0] has unlocked itself.
Pool thread [0] is waiting for a PL API call to serve.
...Pl port listener thread has received a request.
...Pl port listener thread is searching a thread in the pool to serve the request.
......Pl port listener thread is trying to lock pool thread [0].
......Pl port listener thread has successfully locked pool thread [0].
...Pl port listener thread has triggered pool thread [0].
...Pl port listener thread is accepting connections.
Pool thread [0] is serving a PL API call.
...Pool thread [0] has received...
......Pool thread [0]: Bytes in full message: [38]d - [26]h.
......Pool thread [0]: Bytes in message (skipping size header): [34]d - [22]h.
......Pool thread [0]: 22 00 00 00 05 90 4D A5 D1 26 A5 BD 40 85 47 6A F8 CD 07 DA 18
00 00 00 00 01 00 00 00 FF FF FF FF FF FF FF FF 0D
......Pool thread [0]: xx xx xx xx 05 90 4D A5 D1 26 A5 BD 40 85 47 6A F8 CD 07 DA 18
00 00 00 00 01 00 00 00 FF FF FF FF FF FF FF FF 0D
......Pool thread [0]: Op code = [PL_PROTOCOL_OPCODE_WRITE].
......Pool thread [0]: uuid = [d1a54d90-a526-40bd-8547-6af8cd07da18].
......Pool thread [0]: pld = [0].
```

```
......Pool thread [0]: counter offset = [1].
......Pool thread [0]: counter value = [18446744073709551615].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is sending...
......Pool thread [0]: Bytes in full message: [9]d - [9]h.
......Pool thread [0]: Bytes in message (skipping size header): [5]d - [5]h.
......Pool thread [0]: 05 00 00 00 00 00 00 00 0D
......Pool thread [0]: xx xx xx xx 00 00 00 00 0D
......Pool thread [0]: Status = [PL_PROTOCOL_SUCCESS].
......Pool thread [0]: Answer to op code = [PL_PROTOCOL_OPCODE_WRITE].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is closing IPC.
...Pool thread [0] has unlocked itself.
Pool thread [0] is waiting for a PL API call to serve.
...Pl port listener thread has received a request.
...Pl port listener thread is searching a thread in the pool to serve the request.
......Pl port listener thread is trying to lock pool thread [0].
......Pl port listener thread has successfully locked pool thread [0].
...Pl port listener thread has triggered pool thread [0].
...Pl port listener thread is accepting connections.
Pool thread [0] is serving a PL API call.
...Pool thread [0] has received...
......Pool thread [0]: Bytes in full message: [30]d - [1e]h.
......Pool thread [0]: Bytes in message (skipping size header): [26]d - [1a]h.
......Pool thread [0]: 1A 00 00 00 04 90 4D A5 D1 26 A5 BD 40 85 47 6A F8 CD 07 DA 18
00 00 00 00 01 00 00 00 0D
......Pool thread [0]: xx xx xx xx 04 90 4D A5 D1 26 A5 BD 40 85 47 6A F8 CD 07 DA 18
00 00 00 00 01 00 00 00 0D
......Pool thread [0]: Op code = [PL_PROTOCOL_OPCODE_READ].
......Pool thread [0]: uuid = [d1a54d90-a526-40bd-8547-6af8cd07da18].
......Pool thread [0]: pld = [0].
......Pool thread [0]: counter offset = [1].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is sending...
......Pool thread [0]: Bytes in full message: [17]d - [11]h.
......Pool thread [0]: Bytes in message (skipping size header): [13]d - [d]h.
......Pool thread [0]: 0D 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF 0D
......Pool thread [0]: xx xx xx xx 00 00 00 00 FF FF FF FF FF FF FF FF 0D
......Pool thread [0]: Status = [PL_PROTOCOL_SUCCESS].
......Pool thread [0]: Answer to op code = [PL_PROTOCOL_OPCODE_READ].
......Pool thread [0]: Value = [18446744073709551615].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is closing IPC.
...Pool thread [0] has unlocked itself.
Pool thread [0] is waiting for a PL API call to serve.
...Pl port listener thread has received a request.
...Pl port listener thread is searching a thread in the pool to serve the request.
......Pl port listener thread is trying to lock pool thread [0].
......Pl port listener thread has successfully locked pool thread [0].
...Pl port listener thread has triggered pool thread [0].
...Pl port listener thread is accepting connections.
Pool thread [0] is serving a PL API call.
...Pool thread [0] has received...
......Pool thread [0]: Bytes in full message: [26]d - [1a]h.
......Pool thread [0]: Bytes in message (skipping size header): [22]d - [16]h.
......Pool thread [0]: 16 00 00 00 03 90 4D A5 D1 26 A5 BD 40 85 47 6A F8 CD 07 DA 18
00 00 00 00 0D
......Pool thread [0]: xx xx xx xx 03 90 4D A5 D1 26 A5 BD 40 85 47 6A F8 CD 07 DA 18
00 00 00 00 0D
......Pool thread [0]: Op code = [PL_PROTOCOL_OPCODE_CLOSE].
......Pool thread [0]: uuid = [d1a54d90-a526-40bd-8547-6af8cd07da18].
......Pool thread [0]: pld = [0].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is sending...
......Pool thread [0]: Bytes in full message: [9]d - [9]h.
......Pool thread [0]: Bytes in message (skipping size header): [5]d - [5]h.
......Pool thread [0]: 05 00 00 00 00 00 00 00 0D
......Pool thread [0]: xx xx xx xx 00 00 00 00 0D
......Pool thread [0]: Status = [PL_PROTOCOL_SUCCESS].
......Pool thread [0]: Answer to op code = [PL_PROTOCOL_OPCODE_CLOSE].
......Pool thread [0]: Last byte = [13] - [PL_PROTOCOL_EOR].
...Pool thread [0] is closing IPC.
...Pool thread [0] has unlocked itself.
Pool thread [0] is waiting for a PL API call to serve.
Received agent interrupt request from user [<CTRL>+<C>].
Signal handler is signaling pool thread [0].
Signal handler is signaling the ADMIN port listener thread.
Pool thread [0] was interrupted by user request.
...Signal handler is setting-up IPC (ADMIN port).
Pool thread [0] has ended.
...Signal handler is setting-up socket IPC data (ADMIN port).
...Signal handler is resolving IPC address & port (ADMIN port).
...Signal handler is attempting to connect to ADMIN listener thread (ADMIN port).
Admin port listener thread was interrupted by user request.
Admin port listener thread is tearing-down IPC.
Admin port listener thread has ended.
...Signal handler is sending empty-message to ADMIN port listener thread (ADMIN port).
...Signal handler is disconnecting from ADMIN port listener thread (ADMIN port).
...Signal handler is tearing-down ADMIN IPC.
Signal handler is signaling the PL port listener thread.
...Signal handler is setting-up IPC (PL port).
...Signal handler is setting-up socket IPC data (PL port).
...Signal handler is resolving IPC address & port (PL port).
...Signal handler is attempting to connect to PL listener thread (PL port).
...Signal handler is sending empty-message to PL port listener thread (PL port).
```

```
Pl port listener thread was interrupted by user request.
...Signal handler is disconnecting from PL port listener thread (PL port).
Pl port listener thread is tearing-down IPC.
...Signal handler is tearing-down PL IPC.
Pl port listener thread has ended.
API Use Stats:
...pl_open:   [1] Call(s)
...pl_close:  [1] Call(s)
...pl_attach: [0] Call(s)
...pl_read:   [1] Call(s)
...pl_write:  [1] Call(s)
API Errors Stats:
...pl_open:   [0] Error(s)
...pl_close:  [0] Error(s)
...pl_attach: [0] Error(s)
...pl_read:   [0] Error(s)
...pl_write:  [0] Error(s)
Total error(s) count: [0]
Destroying synchronization objects.
De-allocating thread pool data.
De-initializing Windows socket system.
pl_agent has ended.
```

Listing 41: Log Extract (complete transaction example)

7.3 Network Configuration

When compiled in file system-less mode, the API uses two environment variables to specify the IPV4 address and port number in order to communicate with an agent. These two environment variables are `PL_AGENT_ADDRESS` (the IPV4 address environment variable) and `PL_AGENT_PL_PORT` (the port number environment variable).

7.3.1 IP Address

The IPV4 address environment variable is `PL_AGENT_ADDRESS`. If the variable does not exist, then `PL_DEFAULT_PL_AGENT_ADDRESS` (127.0.0.1) is used.

When the symbol `__PL_EXTRA_INPUT_CHECKS__` is defined, then the IPV4 address is checked to see if it belongs to one of the following classes:

- Class A: 000.000.000.000 to 127.255.255.255
- Class B: 128.000.000.000 to 191.255.255.255
- Class C: 192.000.000.000 to 223.255.255.255
- Class D: 224.000.000.000 to 239.255.255.255
- Class E: 240.000.000.000 to 255.255.255.255

7.3.2 Port Number

The port number environment variable is `PL_AGENT_PL_PORT`. If it does not exist, then `PL_DEFAULT_PL_AGENT_PL_PORT` (49253) is used. When `__PL_EXTRA_INPUT_CHECKS__` symbol is defined, then the port number is checked to be between 1 and 65535.

NOTE

The configuration environment variables are checked each time a call to `pl_open()` is issued. This allows running multiple agents on different addresses and/or ports, providing flexibility to dynamically load-balance a distributed system and enable room for scaling. Because the API can guaranty that no data collision occurs if all PL data are aggregated on a single point, no special care has to be taken about where an agent can be started. The sole requirement is that the system hosting the PL sample agent has network access and a file system. An ad-hoc agent may wave this last requirement if it maintains the PL data in volatile memory, for example, and not permanently in a file system.

NOTE

`productivity_link.h` defines the default environment variable names as the following.

```
PL_DEFAULT_PL_AGENT_ADDRESS_ENVAR_NAME
PL_DEFAULT_PL_PORT_ENVAR_NAME
```

NOTE

An instrumented application compiled with the `__PL_FILESYSTEM_LESS__` symbol behaves as if it were using the API compiled in file system-based mode if the following are true.

- The sample agent is started on a system with an accessible file system.
- The sample agent was started without defining any configuration environment variables.
- The sample agent is started on the same system where the instrumented application runs.

8 Downloading and Running the Software Tester Suite

Currently, downloading and installing the Energy Efficient Performance Module is by invitation only. Go to the URL contained in your invitation email and download the specified zip file.

Copy the zip file to `c:\EEPC` and unzip it. Figure 18 shows the resulting directory structure. The script files expect this directory structure. If you store the files differently, you must edit the scripts.

```
C:\EEPC
├──configuration_files
│  ├──os
│  ├──pl_agent
│  └──power
├──docs
├──MSVCsolutions
│  ├──MSVC2008
│  │  └──popcount
│  │     └──popcount
│  │        └──x64
│  │           └──Release
│  ├──MSVC2010
│  │  └──popcount
│  │     └──x64
│  │        ├──Debug
│  │        └──Release
│  └──MSVC2012_Express
│     └──popcount
│        └──popcount
│           └──x64
│              └──Releas
├──run
│  ├──32
│  ├──64
│  ├──outputs
│  └──scripts
└──src
```

Figure 18: Directory Structure of the Downloaded Zip File

8.1 Run `esrv` with the Unoptimized `popcount`

This example assumes that you want to run `esrv` and collect concurrency information. The script `go64.bat` specifies the concurrency dll, but no concurrency information is logged without specifying the `pid` of the application under test. This application is `popcount`.

First, ensure that the directory `c:\EEPC\run\outputs` is empty because this is where `esrv` places the output files.

Start `esrv`. It's good practice to do this in its own command window. This is so that you can watch the output of `esrv` separately from the other commands. Once `esrv` is running, start `popcount` with the appropriate unoptimized options. Then, obtain `popcount`'s `pid` and communicate that `pid` to `esrv`. Wait until `popcount` completes and then stop `esrv`.

The commands are as follows.

```
del c:\EEPC\run\outputs\*
C:\EEPC\run\scripts>go64
            :
            :

| details.                                            |
+-----------------------------------------------------+
[Thu May 09 23:32:41 2013]-DIAG: ......Initializing Power Management Processing. [POWER
MANAGEMENT]
```

```
Samples:        [4]
```

In another command window, start up `popcount`. Note the `--delay` option that specifies a 120 second delay. This is so that you have time to look up `popcount`'s `pid` with the task manager.

```
C:\EEPC\MSVCsolutions\MSVC2010\popcount\x64\Release>popcount --delay 120 --metrics
-parallel 1 --bytes 9000000 --iterations 2000 –runs 1
-----------------------------------------------
Start delay (in s):..................[120]
Problem size (in byte(s)):...........[9000000]
Iteration(s) count:..................[2000]
Parallel processing:.................[YES]
Worker thread(s) count:..............[1]
Metrics requested:...................[YES]
Metrics sampling interval (in ms):...[1000]
Processing mode:.....................[TRIVIAL]
Processing function address:.........[000000013F8D2B70]
:
:
RUN_ID,BUILD,ADDRESSING,THREADS_COUNT,METRICS,METRICS_UPDATE_IN_MS,ALGORITHM,RUN_#,THREAD_
#,START_BYTE,END_BYTE,TOTAL_BYTES,PAYLOAD_BYTES,BITS_SCANNED,BITS_SET,TIME_IN_S
```

Find `popcount`'s `pid`. For this example it is 8656.

```
C:\EEPC\run\64>esrv --device_control pids 8656
```

Watch to see that the `–-device_control` is processed. You may have to issue the above command more than once when running under Microsoft* Windows 7. This has not been necessary under Microsoft* Windows 8.

Wait for `popcount` to finish.

```
RUN_ID,BUILD,ADDRESSING,THREADS_COUNT,METRICS,METRICS_UPDATE_IN_MS,ALGORITHM,RUN_#,THREAD_
#,START_BYTE,END_BYTE,TOTAL_BYTES,PAYLOAD_BYTES,BITS_SCANNED,BITS_SET,TIME_IN_S
08a477be-ee27-44b0-bb74-35395b8e8303,RELEASE,64-
bit,1,YES,1000,TRIVIAL,1,0,0,8999999,9000000,9000000,144000000000,72000000000,53.649982
08a477be-ee27-44b0-bb74-35395b8e8303,RELEASE,64-
bit,1,YES,1000,TRIVIAL,*,*,*,*,*,*,144000000000,72000000000,55.668326

-----------------------------------------------

            :
            :


[Thu May 09 23:49:50 2013]-DIAG: Processing DEVICE CONTROL Command. [ML-COMMAND]
[Thu May 09 23:49:50 2013]-DIAG: ...Signaling DEVICE CONTROL Interrupt. [ML-COMMAND]
            :
            :
```

Stop `esrv`. Watch the command window in which you started `esrv` so that you can see `esrv` stop. Sometimes you have to issue the stop more than once under Microsoft* Windows 7. Sometimes you might even have to issue the stop several times. This has not been necessary under Microsoft* Windows 8.

```
C:\EEPC\run\64>esrv –stop
            :
            :
+----------------------------------------------------------+
| IECSDK Energy Server: STOP                               |
+----------------------------------------------------------+
[Thu May 09 23:31:14 2013]-DIAG: Stopping Energy Server. [COMMON]
```

```
C:\EEPC\run\scripts>
```

Check that you have output files.

```
C:\EEPC\run\outputs>dir /B
test-000000.csv
test_key-000000.csv
```

Look at the output with a Microsoft* Excel. Specifically look at `T(0)` in `test-000000.csv`. `T(0)` has nonzero values if `esrv` successfully collected concurrency data. Figure 19 shows some `T(0)` data.

0
0
0
20380704
20380704
20380704
20380704
20380704
20380704
20380704

Figure 19: `T(0)` Data in `test-000000.csv`

Figure 20 shows the line in `test_key-000000.csv` that identifies `T(0)`.

key	T(0)	Cycles for Proccess [0]:[00006672d - 0x00001a10h], Thread [0]:[00005836d - 0x000016cch].

Figure 20: Definition of `T(0)` in `test_key-000000.csv`

8.2 Run `esrv` with the Optimized `popcount`

The procedure for running the optimized example is similar to that for running the unoptimized version.

There is no need to empty the `outputs` directory. `esrv` increments the number in the CSV files so that there is no conflict.

Start `esrv` in the same way. The switches on the `popcount` invocation are different because they specify an optimized version. The `popcount` invocation is as follows.

```
popcount --delay 120 --metrics --hardware --parallel 2 --bytes 9000000 --iterations 2000 --runs 1
```

As with the unoptimized example, you may have to issue the `esrv` command with `-device_control` more than once. You may also have to issue the `esrv` command with `--stop` more than once.

When `esrv` stops, the `outputs` directory contains the following files.

```
C:\EEPC\run\outputs>dir /B
test-000000.csv
test-000001.csv
test_key-000000.csv
test_key-000001.csv
```

Notice the increment in the file name. The files `test-000001.csv` and `test_key-000001.csv` are the output files for the second run, in this case, the optimized run.

Figure 21 shows how a Win7 desktop looks for the optimized run.

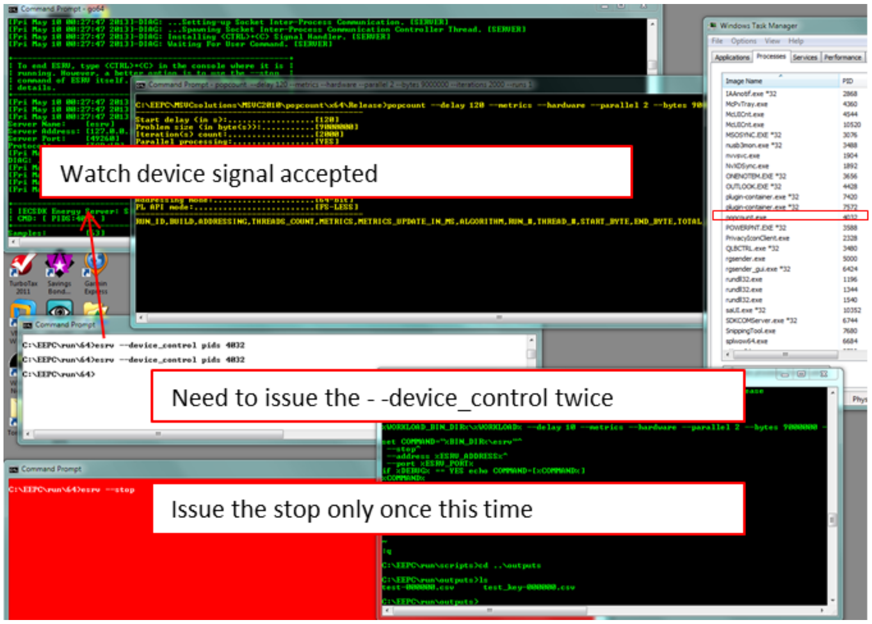


Figure 21: Run `esrv` with the Optimized `popcount`

9 For More Information

Both `esrv` and `popcount` display online help. To see the help for `esrv`, type

```
C:\EEPC\run\64>esrv --help
Start and control energy server.
Usage:  esrv [ --start | --stop | --reset | --help | --ranges |
               --version | --device_control | --status ]
Context-sensitive help is available for each command, i.e., "esrv --start --help"
```

To see the help for `popcount`, type

```
C:\EEPC\MSVCsolutions\MSVC2010\popcount\x64\Release>popcount --help
Counts the number of bits set to one in a stream of bits.

 Usage: POPCOUNT [--DELAY <d>] [--BYTES <b>] [--ITERATIONS <i>]
                 [--PARALLEL <t>] [--METRICS] [--METRICS_UPDATE <s>]
                 [--INTRINSIC | --LIBRARY | --HARDWARE]
                 [--HELP]

 --DELAY <d>:
       <d> Specifies the delay in seconds before starting processing.
       1 s by default. Must be less than 120 s.
 --BYTES <b>:
       <b> Specifies the size of the bits stream in byte(s).
       10003000 bytes by default. Must be less than 100000000 bytes.
 --ITERATIONS <i>:
       <i> Specifies the number of itearation(s) of the processing.
       1500 iteration(s) by default. Must be less than 10000 iteration(s).
 --PARALLEL <t>:
       <t> Specifies the number of worker thread(s) used for processing.
       8 thread(s) by default. Must be less than 24 threads. if --PARALLEL
       is not specified, one worker thread is used for serial processing.
       Serial processing is the default execution mode. Using threads
```

```
        speeds-up the processing.
  --METRICS:
        Activate the metrics thread so counters are exposed. The following
        metrics are exposed:
           - run.
           - total bits checked.
           - total bits found set.
           - bits checked by thread n.
           - bits found set by thread n.
  --METRICS_UPDATE <s>:
        <s> Specifies the counters exposing interval in ms.
        1000 ms by default. Must be mores than 500 ms. if --METRICS_UPDATE
        is specified, --METRICS is assumed automatically.
  --INTRINSIC | --LIBRARY | --HARDWARE:
        Use one of these options - exclusive - to specify the algorithm
        to be used during processing. --TRIVIAL is used by default.
        --INTRINSIC: uses compiler intrinsics to speed-up serial processing.
        --LIBRARY: uses optimized code (library) to speed-up serial processing.
        --HARDWARE: uses hardware acceleration to speed-up serial processing.

 Examples:
        POPCOUNT
        POPCOUNT --DELAY 30
        POPCOUNT --BYTES 50000 --ITERATIONS 2000 --PARALLEL 4 --HARDWARE
        POPCOUNT --METRICS --METRICS_UPDATE 6000
```

The following book is also useful.

> Energy Aware Computing, Powerful Approaches for Green System Design, by Abhishek R. Agrawal, Bob Steigerwald, Chakravarthy Akella, and Chris D. Lucero, available here: http://noggin.intel.com/intelpress/categories/books/energy-aware-computing .