

Linaro Custom Metric Plugin Interface

Release 26.1

Linaro Limited

Nov 05, 2026

Copyright © 2023-2026 Linaro Limited. All rights reserved.
Copyright © 2017-2023 Arm Limited (or its affiliates). All rights reserved.
Copyright © 2002-2017 Allinea Software Limited.

CONTENTS

1 Summary **3**
- 1.1 Documentation . . . 3
- 1.2 Advice to metric authors . . . 3
- 1.3 Advice to profiler authors . . . 4
- 1.4 Static linking . . . 4
  - 1.4.1 Implementing the API . . . 4
- 1.5 Related Pages . . . 5

2 Metric Definition **7**
- 2.1 Metric Definition File . . . 7
  - 2.1.1 `<metric>` . . . 7
  - 2.1.2 `<metricGroup>` . . . 9
  - 2.1.3 `<source>` . . . 10

3 Linaro Performance Reports Integration **11**
- 3.1 Default definition file location . . . 11
- 3.2 Custom definition file location . . . 11
- 3.3 Partial report definition file . . . 11
  - 3.3.1 `<partialReport>` . . . 12
  - 3.3.2 `<reportMetrics>` . . . 13
  - 3.3.3 `<reportMetric>` . . . 13
  - 3.3.4 `<sourceDetails>` . . . 13
  - 3.3.5 `<subsections>` . . . 14
  - 3.3.6 `<subsection>` . . . 14
  - 3.3.7 `<text>` . . . 14
  - 3.3.8 `<entry>` . . . 14
- 3.4 Color codes . . . 14
- 3.5 Reserved Names/IDs and Restrictions . . . 15
- 3.6 HTML Markup . . . 15

4 Quick Start **17**
- 4.1 Get started using Custom Metrics . . . 17
  - 4.1.1 About this task . . . 17
  - 4.1.2 Procedure . . . 17

5 Metric plugin API **19**
- 5.1 System info functions . . . 19
- 5.2 Error reporting functions . . . 20
- 5.3 Memory management functions . . . 22
- 5.4 Standard utility functions . . . 24

6 Metric Plugin Template **29**
6.1 Function documentation . . . 29

7 File List **35**
7.1 Globals . . . 35
7.1.1 Functions and typedefs . . . 35
7.2 Include . . . 36
7.2.1 allinea_metric_plugin_api.h File Reference . . . 37
7.2.2 allinea_metric_plugin_errors.h File Reference . . . 39
7.2.3 allinea_metric_plugin_template.h File Reference . . . 41
7.2.4 allinea_metric_plugin_types.h File Reference . . . 42
7.2.5 allinea_safe_malloc.h File Reference . . . 44
7.2.6 allinea_safe_syscalls.h File Reference . . . 46

8 Examples **49**
8.1 backfill1.c . . . 49
8.2 `backfill1.xml` . . . 50
8.3 custom1.c . . . 50
8.4 `custom1.xml` . . . 52
8.5 `report.xml` . . . 54

9 Proprietary notice **55**

Welcome to the documentation for the Linaro MAP Metric Plugin Interface. The Linaro MAP Metric Plugin Interface enables metric plugin libraries to be written and compiled as a small shared library. This library can then used by Linaro MAP and other profilers implementing the Metric Plugin API.

CHAPTER

ONE

SUMMARY

This section introduces the Linaro MAP Metric Plugin Interface.

1.1 Documentation

The documentation of this interface is composed of the following sections:

- *The metric plugin template*
- *The metric plugin API*
- *Metric Definition File*
- *Linaro Performance Reports Integration*

The metric plugin template documentation specifies which functions must be implemented to create a metric plugin library. This consists of one or more metric *getter* functions that return the values of a metric when called, a pair of optional initialization and cleanup functions called when a metric library shared object is loaded or unloaded, and optional routines which are called when the Linaro Forge sampler is initialized and when sampling ends.

The metric plugin API documents the functions that might be used by metric libraries. The implementation of these functions must be provided by any profiler that intends to use metric plugin libraries to obtain data.

Metric plugin libraries are installed into profilers by providing an XML metric definition file describing what metrics are provided by a metric library, and how those metrics are to be used and displayed.

Linaro Performance Reports can be extended with one or more partial report sections, where the metrics to be displayed can be defined by the user, enabling custom metrics to be part of the default `.html` and the `.txt` report files generated by Linaro Performance Reports.

1.2 Advice to metric authors

There are two main issues to keep in mind when writing a metric plugin library:

- Speed
- Async-signal safety

Linaro MAP aims to avoid adding overhead to the runtime of the program that it profiles. The insertion of a comparatively small amount of overhead can get magnified when MPI communications between a large number of processes are taken into account. For this reason, metric **getters** must return values as fast as possible, and Linaro recommends avoiding long-running operations unless in the *allinea_plugin_initialise()* or *allinea_plugin_cleanup()* functions.

Linaro MAP does its sampling from inside a signal handler. This signal might interrupt any operation, including basic C library functions such as `malloc` or `printf`. It is possible for a signal to interrupt an operation in such a way that for the duration of the signal handler some data structures are left in an inconsistent state. If the code in the signal handler then uses such a data structure (for example, makes another `malloc` call) then the program could deadlock, crash, or otherwise behave in an unpredictable way. To prevent this, code called in a metric plugin `getter` method must avoid making any function calls that are not async-signal safe (allocating memory being the prime example). For convenience, the *Metric plugin API* includes versions of many common functions that are safe to use inside signal handlers (and subsequently, from metric *getter* functions).

See the example metric plugin implementation *custom1.c* and the corresponding definition file *custom1.xml*.

1.3 Advice to profiler authors

To profile using metric plugin libraries, ensure your profiler is setup to:

- Implement all the functions specified in the *Metric plugin API*.
- Parse the metric definitions XML files from an established location.
- Load the shared libraries as described in the <source> elements of those XML files.
- When each metric library is loaded, call the *allinea_plugin_initialise()* function. When each library is unloaded, call its *allinea_plugin_cleanup()* function.
- To obtain values, call the metric *getter* methods (as declared in the metric definitions XML defined in the metric plugin library).
- Normalize, with respect to time, the values obtained from any metric configured with a `divideBySampleTime` attribute set to `true` in their XML definition (see `<metric>` in *Metric Definition File*).
- Store, process, and display the values obtained from the metric plugin libraries, as appropriate.

1.4 Static linking

Custom metrics are not supported in Linaro MAP and Linaro Performance Reports when the Linaro Forge sampler is statically linked.

1.4.1 Implementing the API

Many of the *Metric plugin API* functions are provided for convenience to make async-signal safety less troublesome.

If your profiler never makes metric `getter` calls from signal handlers, but instead always calls them from well-defined (safe) points in user-code, then your API implementation can pass the calls to the `libc` functions (such as *allinea_safe_malloc()* -> `malloc`.

Similarly I/O related utility functions, such as *allinea_safe_read()* and *allinea_safe_write()*, are provided for I/O metric count correctness. If your profiler does not track I/O, then those functions can similarly pass the calls to the corresponding `libc` implementation.

1.5 Related Pages

Here is a list of all related documentation pages:

- *Metric Definition File*

  To add metric libraries that implement the *Metric Plugin Template* and make calls to the *Metric plugin API* to compatible profilers, use a metric definition file

- *Linaro Performance Reports Integration*
- *Quick Start*

  Follow these instructions for the quickest way to get started using custom metrics with Linaro MAP and Linaro Performance Reports.

CHAPTER

TWO

METRIC DEFINITION

This section describes the metric definition file.

2.1 Metric Definition File

This topic describes the Metric Definition File.

To add metric libraries that implement the *Metric Plugin Template* and make calls to the *Metric plugin API* to compatible profilers, use a `metric definition` file.

This is a short `xml` file that defines the location of a metric plugin library and lists both the metrics that library can provide, and how the profiler must process and display the metric data returned by that library.

A metric definition file uses the format:

```
<metricdefinition version="1">
    <!-- 'metric' element(s) defining the metrics provided by the library -->
    <!-- 'metricGroup' element(s) describing how the metrics should be
         grouped when displayed by a profiler -->
    <!-- 'source' element(s) specifying the location of the metrics library
         the profiler should load and, optionally, a list of libraries to
         preload into the profiled application -->
</metricdefinition>
```

See *custom1.xml* for a complete metric definition file. The elements are described in detail here:

2.1.1 `<metric>`

```
<metric id="com.allinea.metrics.myplugin.mymetric1">
    <enabled>[always|default_yes|default_no|never]</enabled>
    <units>%</units>
    <dataType>[uint64_t|double]</dataType>
    <domain>time</domain>
    <onePerNode>[false|true]</onePerNode>
    <source ref="com.allinea.metrics.myplugin_src" functionName="mymetric1" customData="mydata"/>
    <display>
        <description>Human readable description</description>
        <displayName>Human readable display name</displayName>
        <type>instructions</type>
        <colour>green</colour>
    </display>
</metric>
```

Each <metric> element describes a single metric that is provided by a metric plugin library. The id attribute of the opening <metric> element is the identifier used by this definition file and the profiler to uniquely identify this metric. To avoid confusion, ensure that metric element does not contain any whitespace and that it is chosen to minimize the risk of clashing with an existing metric name. To avoid metric name clashes, derive your metric ids from your website domain name and the name of the plugin library, for example com.allinea.metrics.myplugin.mymetric1.

enabled
Specifies whether the metric is enabled or not. Options are always, never, default_yes, default_no enabled. This allows you to explicitly enable or disable the metrics listed as default_no or default_yes enabled using the command line. A metrics source library is not loaded if all metrics which it defines have been disabled.

units
The units this metric is measured in. The profiler might automatically rescale the units to better display large numbers, for example, convert B to KB). Possible values include % (percentage in the range 0..100), B (bytes), B/s (bytes per second), calls/s (calls per second), /s (per second), ns (nanoseconds), J (joule) and W (watts, joules per second). Other units can be specified but the profiler, might not know how to rescale for particularly large or small numbers.

dataType
The datatype used to represent this metric. Possible values are uint64_t (exact integer) and double (floating point).

domain
The domain in which sampling occurs. The only supported domain is time.

onePerNode
If false, all processes report this metric. If true, only one process on each node (machine) calculates and returns this metric. Use true when the metric is a machine-level metric that can not be attributed to an individual process. The default value (if this element is omitted) is false.

If all the metrics of the library have <onePerNode> set to true, only the library is enabled in one process of the node, and this process is the only one to call the allinea_plugin_initialize and allinea_plugin_cleanup functions.

backfill
If true, the metric getter is called once for each sample when the user application is ending (for example, in MPI_Finalize or atexit) or the sampling has stopped after a timeout. If false, or the tag is not present, then the metric getter collects data at sample time. For more information on backfilling, see the examples *backfill1.c* and *backfill1.xml*

source

- **ref**
  The id of the <source> element detailing the metric plugin library that contains that function. The function in functionName must have the appropriate signature for a metric getter, see *mymetric_getIntValue()* as an example.
- **functionName**
  The name of the function to call to obtain a value for this metric.
- **divideBySampleTime (Not used in example)**
  If true, the metric getter function returns the difference in the measured value since the last sample. The profiler divides the returned value by the time elapsed since the last sample to get the true value. If false, the value returned by the getter function is left unaltered.

  As a special case, if units is % (percentage) and divideBySampleTime is true, the result is also multiplied by 100.0 to give a percentage value in the range 0..100.

In this situation the function specified by `functionName` returns a time-in-seconds value.

- **customData**
  Custom data string associated with this metric id, which can be extracted in the metric plugin library with a call to *allinea_get_custom_data*.

- **display**
  Options describing how this metric must display:

  - **description**
    A human-readable description of this metric, suitable for tooltips or short summary text.

  - **displayName**
    The human-readable display name for this metric

  - **type**
    Identifies the broad category this metric falls under. Currently supported values are `cpu_time`, `energy`, `instructions`, `io`, `memory`, `mpi`, and `other`.

  - **colour**
    A color to use when displaying the metric (for example, the color of any graphs generated using this metric). This field is optional. The profiler can choose to use a color based on the `type` field instead of the color code specified here. Color values can be:

    * an RGB hex code of the form #RGB, #RRGGBB, #RRRGGGBBB or #RRRRGGGGBBBB (each of R, G, and B is a single hex digit).
    * an SVG color keyword, for example, `green`.

  - **rel**
    Specifies related metrics. The only supported related metric type is `integral`. This type might be used to specify another metric which is an integral of the metric being defined, for example `rapl_energy` is an integral of `rapl_power`:

    ```
    <rel type="integral" name="rapl_energy"/>
    ```

2.1.2 <metricGroup>

```
<metricGroup id="foo">
    <displayName>Foo Metrics</displayName>
    <description>All metrics relating to foo.</description>
    <metric ref="com.allinea.metrics.myplugin.mymetric1"/>
    <metric ref="com.allinea.metrics.myplugin.mymetric2"/>
    <metric ref="com.allinea.metrics.myplugin.mymetric3"/>
</metricGroup>
```

Each `metricGroup` element describes a collection of metrics to be grouped together. The UI of the profiler might provide actions that apply to all metrics of a group (such as, show or hide all metrics of a group). Metric groups are for display purposes only and do not affect the gathering of metrics.

displayName
A human readable name for this metric group.

description
A human-readable description of this metric group, suitable for tooltips or some short summary text.

metric
One or more elements each referencing a `<metric>` element elsewhere in this `.xml` file.

2.1.3 <source>

Each <source> element represents a metric plugin library that is available for loading. The specific metrics in that library are listed as <metric> elements above.

```
<source id="com.allinea.metrics.myplugin_src">
    <sharedLibrary>lib-myplugin.so</sharedLibrary>
    <preload>lib-mywrapper1.so</preload>
    <preload>lib-mywrapper2.so</preload>
    <functions>
        <start>user_application_start</start>
        <stop>sampling_stop</stop>
    </functions>
</source>
```

id
: The id this library is referenced as in this .xml file. This id is used within the source fields of <metric>.

sharedLibrary
: The library implementing the *Metric Plugin Template*. This must be located in a directory that is checked by the profiler. To resolve this library name, see the documentation of the profiler to determine which directories are searched.

preload
: A list of shared libraries to be preloaded into the application to profile. Preloads are optional, and they might be used to wrap function calls from the profiled application into other shared libraries. The sharedLibrary preloads have to be located in a directory checked by the profiler.

start
: The name of the function to call when the Linaro Forge sampler is initialized. This function is optional but must have the same signature as *start_profiling()*.

stop
: The name of the function to call when the Linaro Forge sampler has ceased taking samples. This function is optional but must have the same signature as *stop_profiling()*.

CHAPTER

THREE

LINARO PERFORMANCE REPORTS INTEGRATION

To integrate one or more metric plugins into Linaro Performance Reports, provide a small partial report file that describes the additions to be made to a Performance Report. This report displays report metrics which are obtained by combining the metric-specified values sampled for a metric. It is possible to select which values from a metric (min, max, mean or sum) to be used and how to accumulate them (min, max or mean).

3.1 Default definition file location

This custom partial report content can be defined in one or multiple custom report definition files, that must be placed under the following folder structure in the default configuration path: `~/.allinea/perf-report/reports/`

When multiple files are found in the directory, all are loaded and used during report generation.

To redefine the default configuration folder, use the `ALLINEA_CONFIG_DIR` environment variable.

Note: To use this feature, the default configuration folders sub-folder structure must be the same (for example, `$ALLINEA_CONFIG_DIR/perf-report/reports/`).

3.2 Custom definition file location

The location of the custom partial report definition files can be overridden using `ALLINEA_PARTIAL_REPORT_SOURCE` environment variable, which can either point to a single XML file or a folder containing XML files. When a folder is specified, the sub-folders are not searched for files but all XML files from that level are loaded in ascending alphabetical order.

3.3 Partial report definition file

General layout of the file is the following: first define the report metrics that need to be used in the report, then define the layout of the subsections that are visible in the report.

An example partial report definition file:

```
<partialReport name="com.allinea.myReport"
            xmlns="http://www.allinea.com/2016/AllineaReports">
<reportMetrics>
```

(continues on next page)

(continued from previous page)

```
    <!-- multiple <reportMetric> elements can be defined -->
    <!-- source attribute must be set to "metric" -->
    <reportMetric id="com.allinea.metrics.sample.average"
                tooltip="Metric tooltip."
                displayName="Average of custom interrupt metric"
                units="/s"
                colour="hsl(19, 70, 71)"
                source="metric">
    <!-- metricRef: reference to an entry in metricdefinitions element -->
    <!-- sampleValue: data value from each sample to take: min, max, mean or sum across processes -->
    <!-- aggregation: how to aggregate per-sample values into a single value: min, max, mean across␣
↪time -->
    <sourceDetails metricRef="com.allinea.metrics.sample.interrupts"
                    sampleValue="mean"
                    aggregation="mean"/>
    </reportMetric>
</reportMetrics>
<subsections>
    <!-- multiple <subsection> elements can be defined -->
    <subsection id="my_metrics"
                heading="My metrics"
                colour="hsl(23, 83, 59)">
    <text>Section one:</text>
    <!-- multiple <entry> elements can be defined -->
    <entry reportMetric="com.allinea.metrics.sample.average"
            group="interruptGroup" />
    </subsection>
</subsections>
</partialReport>
```

3.3.1 <partialReport>

```
<partialReport name="com.allinea.myReport"
          xmlns="http://www.allinea.com/2016/AllineaReports">
</partialReport>
```

name
: A name that uniquely identifies this report. The name is used to distinguish between different kind of reports. See *Reserved Names/IDs and Restrictions* for reserved names.

xmlns
: The `xmlns="http://www.allinea.com/2016/AllineaReports"` attribute must be included in the `<partialReport>` element.

3.3.2 <reportMetrics>

Container for one or more <reportMetric> elements.

3.3.3 <reportMetric>

Each <reportMetric> element describes a single metric that is stored as a double value.

```
<reportMetric id="com.allinea.metrics.sample.average"
displayName="Average of custom interrupt metric" tooltip="Metric
tooltip." units="/s" colour="hsl(19, 70, 71)"
```

id
: A unique identifier for this metric. This is used to reference this metric in <entry> elements. Ids with dots in their value must not be substrings of each other.

  For example, com.allinea.metrics.sample is a substring of com.allinea.metrics.sample.average and therefore not allowed. See *Reserved Names/IDs and Restrictions* for reserved names.

displayName
: The name for this metric, as it appears in the report.

tooltip (optional)
: A tooltip to be displayed when hovering over the metric name.

units
: The units this metric is measured in, as it appears in the report. Units are auto-scaled with SI (G, M, k) and IEC (Gi, Mi, ki) prefixes, but no negative exponent scaling is performed. You must scale custom metrics and select custom units accordingly.

colour (optional)
: The color to be used for this metric when it is named in the report. Color is also used for comparison bars showing the relative value of this metric. See *Color codes*.

source
: Source type of this metric, must be set to metric.

3.3.4 <sourceDetails>

Describes the details of the source metric used in this report element.

```
<sourceDetails metricRef="com.allinea.metrics.sample.interrupts"
sampleValue="mean" aggregation="mean"/>
```

metricRef
: Reference to an existing metric, can be either an Forge-defined or a user-defined (custom) metric.

sampleValue
: Which sample value of the referenced source metric to be used (options: min/max/mean).

aggregation
: How aggregated samples are aggregated (options: min/max/mean).

3.3.5 <subsections>

Container for one or more <subsection> elements.

3.3.6 <subsection>

Describes the layout of the data to be displayed.

```
<subsection id="my_metrics"
           heading="My metrics"
           colour="hsl(23, 83, 59)">
<!-- can add a <text> element with further description -->
<!-- multiple <entry> elements can be defined -->
</subsection>
```

id
: A unique identifier for this subsection.

heading
: The display name for this subsection.

colour (optional)
: The color to use for the heading text. See *Color codes*.

3.3.7 <text>

Short static description of this subsection.

3.3.8 <entry>

Describes one of the metrics to be listed in the subsections of the report.

```
<entry reportMetric="com.allinea.metrics.sample.average"
  group="interruptGroup" />
```

reportMetric
: Referenced report metric that is to be displayed.

group (optional)
: The optional group attribute is used to determine how to scale comparison bars that indicate the relative size of two or more metric values. If two <entry> elements have the same value for their group attribute, they are considered comparable and have comparison bars drawn using the same scale.

3.4 Color codes

Colors can be specified in one of the following forms:

- #RGB (each of R, G, and B is a single hex digit)
- #RRGGBB
- #RRRGGGBBB
- #RRRRGGGGBBBB

- `rgb(X,Y,Z)` (X, Y, and Z are decimal values in the range 0-255)
- `hsv(H,S,V)` (H in the range 0-359, S, and V in the range 0-100)
- `hsl(H,S,L)` (H in the range 0-359, S, and L in the range 0-100)
- A name from the list of colors defined in the list of SVG color keyword names provided by the World Wide Web Consortium: https://www.w3.org/TR/SVG11/types.html#ColorKeywords

3.5 Reserved Names/IDs and Restrictions

All names and IDs starting with `allinea.` or `com.allinea.` are reserved and can not be used in the partial report definition file. Instead, for example, use your reversed Internet domain name as prefix.

IDs must be a valid XML NCName (see https://www.w3.org/TR/xmlschema-2/#NCName) and can not contain symbols (except `.`, `_` and `-`).

Note: IDs can not begin with `.` or `_`.

3.6 HTML Markup

Most text in the partial report definition XML file that are inserted directly into the generated report can contain a limited subset of HTML markup. Supported elements include headings, ordered and unordered lists, such as `span`, `div`, `p`, `a` `b`, `i`, and `img`.

CHAPTER

FOUR

QUICK START

This section describes the how to quickly start using custom metrics with Linaro MAP and Linaro Performance Reports.

4.1 Get started using Custom Metrics

Follow the instructions in this section to get started using custom metrics with Linaro MAP and Linaro Performance Reports.

4.1.1 About this task

The development of custom metrics for use with Linaro MAP requires you to read and understand:

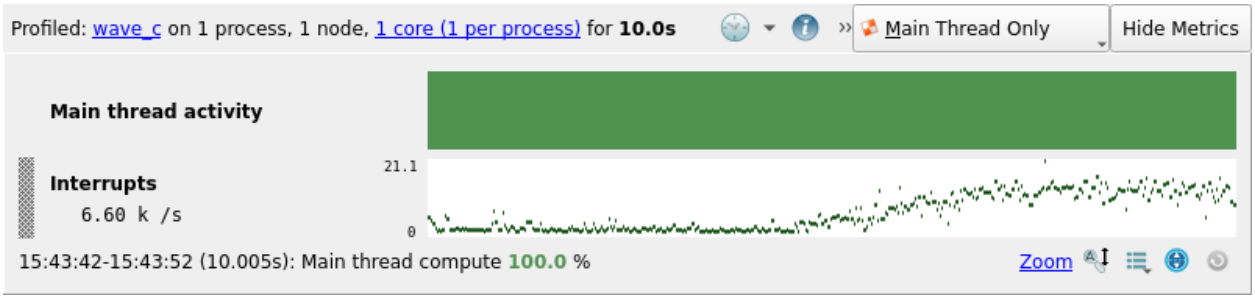
- the *Documentation* section which highlights the common pitfalls when writing custom metrics.
- the *Metric Definition File* section which provides details about the meta information in *custom1.xml* that Linaro MAP requires to run and display the custom metrics.
- the *Metric Plugin Template* section which describes the functions which need to be implemented by a custom metrics library.

In addition, see information on exposing custom metrics in Linaro Performance Reports in *Linaro Performance Reports Integration*.

4.1.2 Procedure

1. Open a terminal in the `/custom/examples/` directory, which contains:
   - a Makefile for building the custom metrics shared library.
   - the source for the example custom metric (*custom1.c*).
   - ***report.xml*, which explains to Linaro Performance Reports**
     how to access the custom metric.
   - ***custom1.xml*, which provides**
     metadata about this metric to Linaro MAP.
2. If a custom configuration directory for the Linaro Forge tool is in use, set the `ALLINEA_CONFIG_DIR` environment variable to the path of the custom configuration directory.
3. To build and install the custom metric library to the default location (or that specified by `ALLINEA_CONFIG_DIR`), run `make` followed by `make install`.

4. Begin profiling an application as normal with Linaro MAP. To display the custom metric upon completion of the run, use the Metrics menu (*Metrics ▸ Preset: Custom1*). Here is an example of how this looks:



In addition, the `.html` and the `.txt` report files generated by Linaro Performance Reports have an extra section containing the custom metric data.

CHAPTER

FIVE

METRIC PLUGIN API

This section describes the API functions available for use by a metric plugin library.

5.1 System info functions

Functions that provide information about the system or the enclosing profiler.

int allinea_get_logical_core_count(void)
Counts the number of logical cores on this system.

This count includes effective cores reported by hyperthreading.

Returns

The number of CPU cores known to the kernel (including those added by hyperthreading). `-1` if this information is not available.

See also `allinea_get_physical_core_count`

int allinea_get_physical_core_count(void)
Counts the number of physical cores on this system.

This count does not include the effective cores that are reported when using hyperthreading.

Returns

The number of CPU cores known to the kernel (excluding those added by hyperthreading). `-1` if this information is not available.

See also `allinea_get_logical_core_count`

int allinea_read_config_file (const char * variable, const char * metricId, char * value, int length)
Reads the configuration file to find the value of a variable.

```
int allinea_read_config_file  ( const char *  variable,
                                const char *  metricId,
                                char *        value,
                                int           length
                              )
```

This function returns the value of a configuration variable, or an error. If the file is empty, the variable is not found or the variable is improperly declared. This function must only be called from outside of the Linaro Forge sampler (such as in `allinea_plugin_initialise` and similar functions) because it is not async signal safe.

Parameters

Direction	Parameter	Description
[in]	variable	The name of the configuration variable.
[in]	metricId	The ID of the metric with the configuration file environment variable.
[out]	value	The value of the configuration variable.
[in]	length	The length of value.

Returns

0 if there are no errors. -1 if the file name is too long. -2 if the file does not exist. -3 if the variable is not found or is improperly declared.

5.2 Error reporting functions

Functions for reporting errors encountered by either a specific metric or an entire metric plugin library.

void allinea_set_plugin_error_message(plugin_id_t_t plugin_id, int error_code, const char * error_message)

Reports an error that occurred in the plugin (group of metrics).

See *plugin_id_t_t*.

```
void allinea_set_plugin_error_message ( plugin_id_t_t     metric_id,
                                        int               error_code,
                                        const char *      error_message
                                      )
```

This method takes a plain text string as its error_message. Instead, use allinea_set_plugin_error_messagef() to include specific details in the string using printf-style substitution.

This method must only be called from within *allinea_plugin_initialise()*, and only if the plugin library is not able to provide its data (for example if the required interfaces are not present or supported by the system).

Parameters

Parameter	Description
plugin_i	The id identifying the plugin that has encountered an error. The appropriate value is previously passed in as an argument to the *allinea_plugin_initialise()* call.
error_c	An error code that can be used to distinguish between the possible errors that might have occurred. The exact value is up to the plugin author but each error condition must have its own unique error code. In the case of a failing libc function, the libc errno (from <errno.h>) might be appropriate, but a plugin-author-specified constant can also be used. Linaro recommends that you document the meaning of the possible error codes for the benefit of users of your plugin.
error_m	A text string describing the error in a human-readable form. In the case of a failing libc function, the value strerror(errno) might be appropriate, but a plugin-author-specified message can also be used.

void allinea_set_plugin_error_messagef (plugin_id_t_t plugin_id, int error_code, const char * error_message,...)

Reports that an error occurred in the plugin (group of metrics).

See *plugin_id_t_t*.

```
void allinea_set_plugin_error_messagef ( plugin_id_t_t        plugin_id,
                                         int                  error_code*,
                                         const char *         error_message*
                                                              ...
                                       )
```

This method does printf-style substitutions to format values inside the error message.

This method must only be called from within *allinea_plugin_initialise()*, and only if the plugin library is not able to provide its data (for example, if the required interfaces are not present or supported by the system).

Parameters

Parameter	Description
plugin_i	The id identifying the plugin that has encountered an error. The appropriate value is previously passed in as an argument to the *allinea_plugin_initialise()* call.
error_c	An error code that can be used to distinguish between the possible errors that might have occurred. The exact value is up to the plugin author but each error condition must have its own unique error code. In the case of a failing libc function, the libc `errno` (from `<errno.h>`) might be appropriate, but a plugin-author-specified constant can also be used. The meaning of the possible error codes must be documented for the benefit of users of your plugin.
error_m	A text string describing the error in a human-readable form. In the case of a failing libc function, the value `strerror(errno)` might be appropriate, but a plugin-author-specified message can also be used. This can include printf-style substitution characters.
...	Zero or more values for substituting into the `error_message` string in the same manner as `printf`.

Examples

See *custom1.c*.

void allinea_set_metric_error_message (metric_id_t_t metric_id, int error_code, const char * error_message)

Reports that an error occurred when reading a metric.

See *metric_id_t_t*.

```
void allinea_set_metric_error_message ( metric_id_t_t     metric_id,
                                        int               error_code,
                                        const char *      error_message
                                      )
```

Parameters

Parameter	Description
metric_id	The id identifying the metric that has encountered an error. The appropriate value is passed in as an argument to the metric `getter` call.
error_code	A pointer to the start of the memory region to free. This must be previously allocated with `allinea_safe_malloc()`, `allinea_safe_realloc()`, or `allinea_safe_calloc()`.
error_messa	A text string describing the error in a human-readable form. In the case of a failing libc function, the value `strerror(errno)` might be appropriate, but a plugin-author-specified message can also be used.

void allinea_set_metric_error_messagef (metric_id_t_t metric_id, int error_code, const char * error_message,...)

Reports that an error occurred when reading a metric.

See *metric_id_t_t*.

```
void allinea_set_metric_error_messagef ( metric_id_t_t    metric_id,
                                         int              error_code,
                                         const char *     error_message
                                                          ...
                                       )
```

This method uses printf-style substitutions to format values inside the error message.

Parameters

Parameter	Direction
metric_i	The id identifying the metric that has encountered an error. The appropriate value is previously passed in as an argument to the metric `getter` call.
error_c	An error code that can be used to distinguish between the possible errors that might have occurred. The exact value is up to the plugin author but each error condition must have its own unique error code. In the case of a failing libc function the libc `errno` (from `<errno.h>`) might be appropriate, but a plugin-author-specified constant can also be used. The meaning of the possible error codes must be documented for the benefit of users of your plugin.
error_m	A text string describing the error in a human-readable form. In the case of a failing libc function, the value `strerror(errno)` might be appropriate, but a plugin-author-specified message can also be used. This can include printf-style substitution characters.
...	Zero or more values to be substituted into the `error_message` string.

Examples

See *custom1.c*.

5.3 Memory management functions

Async signal safe replacements for memory management functions.

Because metric library functions need to be async signal safe, the standard libc memory management functions (such as `malloc`, `free`, `new`, `delete`) cannot be used. The following memory management functions can safely be used by the metric plugin libraries even if they are called from inside a signal handler.

void * allinea_safe_malloc(size_t size)

An async-signal-safe version of `malloc`.

Allocates a memory region of size bytes. To be used instead of the libc `malloc`.

If memory is exhausted, an error is printed to `stderr` and the process is aborted.

Memory allocated by this function must be released by a call to `allinea_safe_free()`.

Do not use the libc `free()` to free memory allocated by this function.

Parameters

Direction	Parameter	Description
[in]	size	The number of bytes of memory to allocate.

Returns

A pointer to the start of the allocated memory region.

void allinea_safe_free(void * ptr)

An async-signal-safe version of `free`.

Frees a memory region previously allocated with `allinea_safe_malloc`. Use this instead of the libc `free`. Do not use this function to deallocate memory blocks previously allocated by the libc `malloc`.

Parameters

Direction	Parameter	Description
`[in, out]`	`ptr`	A pointer to the start of the memory region to free. This must be previously allocated with `allinea_safe_malloc()`, `allinea_safe_realloc()`, or `allinea_safe_calloc()`.

void * allinea_safe_calloc(size_t nmemb, size_t size)

An async-signal-safe version of `calloc`. Allocates `size` and `nmemb` bytes and zero-initializes the memory.

```
void* allinea_safe_calloc ( size_t *nmemb*,
                            size_t *size*
                          )
```

Use this instead of the libc `calloc`.

If memory is exhausted, an error is printed to `stderr` and the process is aborted. Memory allocated by this function must be released by a call to `allinea_safe_free()`.

Do not use libc `free` to free memory allocated by this function.

Parameters

Direction	Parameter	Description
`[in]`	`nmemb`	The number of bytes per element to allocate.
`[in]`	`size`	The number of elements to allocate.

Returns

A pointer to the start of the allocated memory region.

int allinea_safe_close (int fd)

Closes the file descriptor `fd` previously opened by `allinea_safe_open(async-signal-safe)`. A replacement for `close`.

When used in conjunction with `allinea_safe_read()` and `allinea_safe_write()`, the bytes read or bytes written are not included in the I/O accounting of the enclosing profiler.

Parameters

Parameter	Description
`fd`	The file descriptor to close.

Returns

`0` on success; `-1` on failure and `errno set`.

Examples

See *custom1.c*.

void * allinea_safe_realloc(void * ptr, size_t size)

An async-signal-safe version of `realloc`.

Reallocates a memory region if necessary, or allocates a new one if `NULL` is supplied for `ptr`.

```
void* allinea_safe_realloc ( void  *  ptr,
                             size_t   size
                           )
```

Use instead of the libc `realloc`.

If memory is exhausted, an error is printed to `stderr` and the process is aborted.

Pointers to memory regions supplied to this function must be allocated by a call to `allinea_safe_malloc()`, `allinea_safe_calloc()` or `allinea_safe_realloc()`.

Memory allocated by this function must be released by a call to `allinea_safe_free()`.

Do not use libc `free` to free memory allocated by this function.

Parameters

Direction	Parameter	Description
`[in]`	`ptr`	The starting address of the memory region to reallocate.
`[in]`	`size`	The new minimum size to request.

Returns

A pointer to a memory region with at least `size` bytes available.

5.4 Standard utility functions

Replacements for common libc utility functions.

Since metric library functions need to be async signal safe most standard libc functions can not be used. In addition, even basic syscalls (such as `read` and `write`) cannot be used without risking corruption of some other metrics that the enclosing profiler might be tracking (for example, bytes read or bytes written). The following functions can be safely called inside signal handlers and accommodates I/O being done by the metric plugin without corrupting I/O metrics that are tracked by the enclosing profiler.

`struct timespec allinea_get_current_time(void)`

Gets the current time using the same clock as the enclosing profiler (`async-signal-safe`).

A replacement for `clock_gettime` that uses the enclosing profiler-preferred system clock (for example, `CLOCK_MONOTONIC`).

Returns

The current time.

Examples

See *custom1.c*.

`const char * allinea_get_custom_data (metric_id_t_t metricId)`

Returns the `customData` attribute of the source element from the metric definition defined in the xml file. See *metric_id_t_t*.

Parameters

Parameter	Description
`metricId`	The metric id.

Returns

The custom data for the given metric id. A zero length C string if not available.

void allinea_safe_fprintf(int fd, const char * format,...)
An async-signal-safe version of `fprintf`.

```
void allinea_safe_fprintf ( int          fd,
                            const char * format,
                                         ...
                          )
```

Parameters

Parameter	Description
`fd`	The file descriptor to write to.
`format`	The format string.
`...`	Zero or more values to be substituted into the `format` string in the same manner as `printf`.

int allinea_safe_open(const char * file, int oflags,...)
Opens the given **file** for reading or writing (async-signal-safe).

```
int allinea_safe_open ( const char * file,
                        int          oflags,
                                     ...
                      )
```

A replacement for `open`. When used in conjunction with `allinea_safe_read()` and `allinea_safe_write()`, the bytes read or bytes written is not included in the I/O accounting of the enclosing profiler.

Parameters

Parameter	Description
`file`	The name of the file to open (might be an absolute or a relative path).
`oflags`	Flags specifying how the file must be opened. Accepts all the flags that can be given to the libc `open` function, such as `O_RDONLY`, `O_WRONLY`, or `O_RDWR`.

Returns

The file descriptor of the open file; `-1` on failure and `errno` set.

Examples

See *custom1.c*.

void allinea_safe_printf(const char * format,...)
An async-signal-safe replacement for `printf`.

```
void allinea_safe_printf ( const char * format,
                                        ...
                         )
```

Parameters

Parameter	Description
`format`	The format string.
`...`	Zero or more values to be substituted into the `format` string in the same manner as `printf`.

ssize_t allinea_safe_read(int fd, void * buf, size_t count)

Reads up to count bytes from buf to fd (async-signal-safe).

```
ssize_t allinea_safe_read ( int     fd,
                            void *  buf,
                            size_t  count
                          )
```

A replacement for read. When used in conjunction with allinea_safe_open() and allinea_safe_close(), the read bytes are excluded from the enclosing profiler's I/O accounting.

Parameters

Parameter	Description
fd	The file descriptor to read from.
buf	The buffer to read to.
count	The maximum number of bytes to read.

Returns

The number of bytes actually read; -1 on failure and errno set.

ssize_t allinea_safe_read_all(int fd, void * buf, size_t count)

Reads the entire contents of fd into buf (async-signal-safe).

```
ssize_t allinea_safe_read_all ( int     fd,
                                void *  buf,
                                size_t  count
                              )
```

When used in conjunction with allinea_safe_open() and allinea_safe_close(), the read bytes are excluded from the enclosing profiler's I/O accounting.

Parameters

Parameter	Description
fd	The file descriptor to read from.
buf	Buffer in which to copy the contents.
count	Size of the buffer. At most, this many bytes are written to buf.

Returns

If successful, shows the number of bytes read, otherwise -1 and errno is set.

ssize_t allinea_safe_read_all_with_alloc(int fd, void * buf, size_t * count)

Reads the entire contents of fd into buf (async-signal-safe).

```
ssize_t allinea_safe_read_all_with_alloc ( int        fd,
                                           void   **  buf,
                                           size_t *   count
                                         )
```

When used in conjunction with allinea_safe_open() and allinea_safe_close(), the read bytes are excluded from the enclosing profiler's I/O accounting.

When this is no longer required, use allinea_safe_free() to allocate sufficient space for the file contents and add a terminating NULL.

Parameters

Parameter	Description
fd	The file descriptor to read from.
buf	The pointer to when the buffer pointer must be stored.
count	Size of the buffer allocated.

Returns

If successful, shows the the number of bytes read, otherwise -1 and errno is set.

ssize_t allinea_safe_read_line(int fd, void * buf, size_t count)

Reads a line from fd into buf (async-signal-safe).

```
ssize_t allinea_safe_read_line ( int    fd,
                                 void *  buf,
                                 size_t  count
                               )
```

The final newline \\n is removed and a final \\0 added. When used in conjunction with allinea_safe_open() and allinea_safe_close(), the written bytes are excluded from the enclosing profiler's I/O accounting.

Lines longer than count are truncated.

Parameters

Parameter	Description
fd	The file descriptor to read from.
buf	Buffer in which to copy the contents.
count	Size of the buffer. At most, this many bytes are written to buf.

Returns

If successful, shows the number of bytes read, otherwise -1 and errno is set.

Examples

See *custom1.c*.

void allinea_safe_vfprintf(int fd, const char * format, va_list ap)

An async-signal-safe version of vfprintf.

```
void allinea_safe_vfprintf ( int           fd,
                             const char *   format,
                             va_list        ap
                           )
```

Parameters

Parameter	Description
fd	The file descriptor to write to.
format	The format string.
ap	A list of arguments for **format**.

ssize_t allinea_safe_write(int fd, const void * buf, size_t count)

Writes up to count bytes from buf to fd (async-signal-safe).

```
ssize_t allinea_safe_write ( int          fd,
                             const void *  buf,
                             size_t        count
                           )
```

A replacement for `write` When used in conjunction with `allinea_safe_open()` and `allinea_safe_close()`, the written bytes are excluded from the I/O accounting of the enclosing profiler.

Parameters

Parameter	Description
`fd`	The file descriptor to write to.
`buf`	The buffer to write from.
`count`	The number of bytes to write.

Returns

The number of bytes actually written; `-1` on failure and `errno` set.

CHAPTER

SIX

METRIC PLUGIN TEMPLATE

This section describes the functions that must be implemented by every metric plugin library.

A metric plugin library is a small shared library that implements the functions `allinea_plugin_initialise()` and `allinea_plugin_clean()`, and is called when the shared library is loaded or unloaded. It also implements one or more functions of the **form** (but not necessarily of the same function name) as `mymetric_getIntValue()` or `mymetric_getDoubleValue()`.

See *custom1.c* for an example of a metric plugin that implements this template.

See *Metric plugin API* for the functions that can be called by this metric library.

See *Metric Definition File* for information on the format of the definition file that informs profilers what metrics the metric plugin library can provide.

6.1 Function documentation

`int allinea_plugin_cleanup(plugin_id_t plugin_id, void * data)`

Cleans a metric plugin being unloaded. See *plugin_id_t*

```
int allinea_plugin_cleanup ( plugin_id_t plugin_id,
                             void *       data
                           )
```

This function must be implemented by each metric plugin library. It is called when that plugin library is unloaded. Use this function to release any held resources (open files etc). Unlike most functions used in a metric plugin library, this is *not* called from a signal handler. Therefore, it is safe to make general function calls and even allocate or deallocate memory using the normal libc malloc/free new/delete functions.

Note: This is called after metric data has been extracted and transferred to the frontend. Therefore, you can not see plugin error messages set by *allinea_set_plugin_error_message()* or *allinea_set_plugin_error_messagef()*.

Parameters

Parameter	Description
`plugin_id`	Opaque handle for the metric plugin. Use this when making calls to `allinea_set_plugin_error_message()` or `allinea_set_plugin_error_messagef()`.
`data`	Currently unused, is always NULL.

Returns

`0` on success; `-1` on error. A description of the error must be supplied using `allinea_set_plugin_error_message()` or `allinea_set_plugin_error_messagef()` before returning.

Examples:

- *backfill1.c*
- *custom1.c*.

Related information

- *allinea_set_plugin_error_message()*
- *allinea_set_plugin_error_messagef()*

`int allinea_plugin_initialise(plugin_id_t plugin_id, void * data)`

Initializes a metric plugin. See *plugin_id_t*

```
int allinea_plugin_initialize ( plugin_id_t plugin_id,
                                void *      data
                              )
```

This function must be implemented by each metric plugin library. It is called when that plugin library is loaded. Use this function to setup data structures and do one-off resource checks. Unlike most functions used in a metric plugin library this is *not* called from a signal handler. Therefore, it is safe to make general function calls and allocate or deallocate memory using the normal libc malloc/free new/delete functions.

If it can be determined that this metric plugin cannot function, for example, the required information is not available on this machine, then it must call `allinea_set_plugin_error_message()` or `allinea_set_plugin_error_messagef()` to explain the situation, and return -1.

Parameters

Parameter	Description
`plugin_id`	Opaque handle for the metric plugin. Use this when making calls to `allinea_set_plugin_error_message()` or `allinea_set_plugin_error_messagef()`
`data`	Currently unused, is always be NULL.

Returns

0 on success; -1 on error. A description of the error must be supplied using `allinea_set_plugin_error_message()` or `allinea_set_plugin_error_messagef()` before returning.

Examples:

- *backfill1.c*
- *custom1.c*.

Related information

- *allinea_set_plugin_error_message()*
- *allinea_set_plugin_error_messagef()*

`int mymetric_getDoubleValue(metric_id_t id, struct timespec * currentSampleTime, double * outValue)`

Example of a floating-point metric `getter` function. See *metric_id_t*.

```
int mymetric_getDoubleValue ( metric_id_t       id,
                              struct timespec * currentSampleTime,
                              double *          outValue
                            )
```

An example of a `getter` function that returns a floating point metric value. Real `getter` functions must be registered with the profiler using a *Metric Definition File*. For example, this function (if it existed) would be registered by having a `<metric>` element along the lines of :

```
1 <metric id="com.allinea.metrics.myplugin.mymetric">
2     <units>%</units>
3     <dataType>double</dataType>
4     <domain>time</domain>
5     <source ref="com.allinea.metrics.myplugin_src" functionName="mymetric_getValue"/>
6     <display>
7         <description>Human readable description</description>
8         <displayName>Human readable display name</displayName>
9         <type>instructions</type>
10         <colour>green</colour>
11     </display>
12 </metric>
```

The most relevant line contains `functionName="mymetric_getValue"`. See *Metric Definition File* for more details on the format of this XML file.

Parameters

Direction	Parameter	Description
`[in]`	`id`	An id used by the profiler to identify this metric. This can be used in calls to *Metric plugin API* functions, such as *allinea_set_metric_error_message()*.
`[in, out]`	`currentSampleTime`	The current time. This time is acquired from a monotonic clock which reports the time elapsed from some fixed point in the past. It is unaffected by changes in the system clock. This is passed in from the profiler to avoid unnecessary calls to *allinea_get_current_time()*. If this metric is backfilled then this time is not the current time, instead it is the time at which the sample was taken and the time that the Linaro Forge sampler is now requesting a data point for. This parameter is additionally an `out` parameter and might be updated with the result from a call to `allinea_get_current_time()` to ensure the `currentSampleTime` is close to the point where the metric is read. Updating `currentSampleTime` from any other source is undefined. In the case of a backfilled metric, `currentSampleTime` does not function as an `out` parameter and results in an error if it is used as such. It is safe to assume that this pointer is not NULL.

Parameters

Direction	Parameter	Description
`[out]`	`outValue`	The return value to be provided to the profiler. It is safe to assume that this pointer is not NULL.

Returns

0 if a metric was written to **outValue** successfully, a non-zero value if there was an error. In the case of an error this function must call *allinea_set_metric_error_message()* before returning.

Caution: This function might have been called from inside a signal handler. Implementations must not make calls that are not async-signal safe. Do not use any function that implicitly or explicitly allocates or frees memory, or uses non-reentrant functions, with the exception of the memory allocators provided

by the *Metric plugin API* (for example, *allinea_safe_malloc()* or *allinea_safe_free()*). Failure to observe async-signal safety can result in deadlocks,segfaults, or undefined/unpredictable behavior.

Note: Do not implement this function! Instead implement functions with the same signature but with a more appropriate function name.

`int mymetric_getIntValue(metric_id_t id, struct timespec * currentSampleTime, uint64_t * outValue)`

Example of an integer metric `getter` function.

See *metric_id_t*.

```
int mymetric_getIntValue ( metric_id_t        id,
                           struct timespec * currentSampleTime,
                           uint64_t  *       outValue
                         )
```

An example of a `getter` function that returns an integer metric value. Real `getter` functions must be registered with the profiler using a *Metric Definition File*. For example, this function (if it existed) would be registered by having a `<metric>` element along the lines of this:

```
1 <metric id="com.allinea.metrics.myplugin.mymetric">
2     <units>%</units>
3     <dataType>uint64_t</dataType>
4     <domain>time</domain>
5     <source ref="com.allinea.metrics.myplugin_src" functionName="mymetric_getValue"/>
6     <display>
7         <description>Human readable description</description>
8         <displayName>Human readable display name</displayName>
9         <type>instructions</type>
10         <colour>green</colour>
11     </display>
12 </metric>
```

The most relevant line being the one containing `functionName="mymetric_getValue"`. See *Metric Definition File* for more details on the format of this XML file.

Parameters

`[in]`	`id`	An id used by the profiler to identify this metric. This can be used in calls to *Metric plugin API* functions, such as *allinea_set_metric_error_message()*.
`[in, out]`	`currentSampleTime`	The current time. This time is acquired from a monotonic clock which reports the time elapsed from some fixed point in the past. It is unaffected by changes in the system clock.

This is passed in from the profiler to avoid unnecessary calls to *allinea_get_current_time()*. If this metric is backfilled, then this time is not the current time. Instead it is the time at which the sample was taken and the time the Linaro Forge sampler is now requesting a data point for.

This parameter is additionally an out parameter and can be updated with the result from a call to `allinea_get_current_time()` to ensure the `currentSampleTime` is close to the point where the metric is read. Updating `currentSampleTime` from any other source is undefined. In the case of a backfilled metric, `currentSampleTime` does not function as an out parameter and results in an error if it is used as such. It is safe to assume that this pointer is not NULL.

Parameters

Direction	Parameter	Description
[out]	outValue	The return value to be provided to the profiler. It is safe to assume that this pointer is not NULL.

Returns

0 if a metric was written to outValue successfully, a non-zero value if there was an error. In the case of an error this function must call allinea_set_metric_error_message() before returning.

Warning: This function might have been called from inside a signal handler. Implementations must not make calls that are not async-signal safe. Do not use any function that implicitly or explicitly allocates or frees memory, or uses non-reentrant functions, with the exception of the memory allocators provided by the *Metric plugin API* (such as, *allinea_safe_malloc()* or *allinea_safe_free()*). Failure to observe async-signal safety can result in deadlocks, segfaults or undefined/unpredictable behavior.

Note: Do not implement this function, Instead implement functions with the same signature but with a more appropriate function name.

int start_profiling(plugin_id_t plugin_id)

Called when the Linaro Forge sampler is initialized. See *plugin_id_t*

```
int start_profiling ( plugin_id_t  *plugin_id* )
```

An example of a function which is called when the Linaro Forge sampler is initialized. This callback is optional and does not need to be implemented. If this function exists it can be registered as follows.

```
1 <source id="com.allinea.metrics.backfill_src">
2     <sharedLibrary>libbackfill1.so</sharedLibrary>
3     <functions>
4         <start>start_profiling</start>
5     </functions>
6 </source>
```

This function does not need to be async-signal-safe as it is not called from a signal.

Parameters

Parameter	Description
plugin_id	Opaque handle for the metric plugin. Use this when making calls to *allinea_set_plugin_error_message()* or *allinea_set_plugin_error_messagef()*.

Returns

0 on success; -1 on error. A description of the error must be supplied using allinea_set_plugin_error_message() or allinea_set_plugin_error_messagef() before returning.

Examples:

backfill1.c.

int stop_profiling(plugin_id_t plugin_id)

Called after the Linaro Forge sampler stops sampling. See *plugin_id_t*.

```
int stop_profiling ( plugin_id_t *plugin_id* )
```

An example of a function which is called when the Linaro Forge sampler finishes sampling. This callback is optional and does not need to be implemented. If this function exists, it can be registered as follows:

```
1 <source id="com.allinea.metrics.backfill_src">
2     <sharedLibrary>libbackfill1.so</sharedLibrary>
3     <functions>
4         <start>stop_profiling</start>
5     </functions>
6 </source>
```

Warning: This function can be called from a signal handler, and therefore it must be async-signal-safe

Parameters

Parameter	Description
plugin_id	Opaque handle for the metric plugin. Use this when making calls to *allinea_set_plugin_error_message()* or *allinea_set_plugin_error_messagef()*,

Returns

`0` on success; `-1` on error. A description of the error must be supplied using `allinea_set_plugin_error_message()` or `allinea_set_plugin_error_messagef()` before returning.

Examples:

backfill1.c.

CHAPTER

SEVEN

FILE LIST

This section describes all the documented `allinea_*.h` files, functions, variables, defines, enums, and typedefs.

7.1 Globals

This section describes all the documented functions, variables, defines, enums, and typedefs.

7.1.1 Functions and typedefs

List of global API and template functions, and type definitions.

API functions

See *Metric plugin API*

- `allinea_get_current_time()`: allinea_safe_syscalls.h
- `allinea_get_custom_data()`: allinea_metric_plugin_api.h
- `allinea_get_custom_data()`: allinea_metric_plugin_api.h
- `allinea_get_logical_core_count()`: allinea_metric_plugin_api.h
- `allinea_get_physical_core_count()`: allinea_metric_plugin_api.h
- `allinea_read_config_file()`: allinea_metric_plugin_api.h
- `allinea_safe_calloc()`: allinea_safe_malloc.h
- `allinea_safe_close()`: allinea_safe_syscalls.h
- `allinea_safe_fprintf()`: allinea_safe_syscalls.h
- `allinea_safe_free()`: allinea_safe_malloc.h
- `allinea_safe_malloc()`: allinea_safe_malloc.h
- `allinea_safe_open()`: allinea_safe_syscalls.h
- `allinea_safe_printf()`: allinea_safe_syscalls.h
- `allinea_safe_read()`: allinea_safe_syscalls.h
- `allinea_safe_read_all()`: allinea_safe_syscalls.h

- `allinea_safe_read_all_with_alloc()`: allinea_safe_syscalls.h
- `allinea_safe_read_line()`: allinea_safe_syscalls.h
- `allinea_safe_realloc()`: allinea_safe_malloc.h
- `allinea_safe_vfprintf()`: allinea_safe_syscalls.h
- `allinea_safe_write()`: allinea_safe_syscalls.h
- `allinea_set_metric_error_message()`: allinea_metric_plugin_errors.h
- `allinea_set_metric_error_messagef()`: allinea_metric_plugin_errors.h
- `allinea_set_plugin_error_message()`: allinea_metric_plugin_errors.h
- `allinea_set_plugin_error_messagef()`: allinea_metric_plugin_errors.h

Metric plugin template functions

See *Metric plugin template*

- `allinea_plugin_cleanup()`: allinea_metric_plugin_template.h
- `allinea_plugin_initialise()`: allinea_metric_plugin_template.h
- `mymetric_getDoubleValue()`: allinea_metric_plugin_template.h
- `mymetric_getIntValue()`: allinea_metric_plugin_template.h
- `start_profiling()`: allinea_metric_plugin_template.h
- `stop_profiling()`: allinea_metric_plugin_template.h

Type definitions

See : *allinea_metric_plugin_types.h*

- `metric_id_t`
- `plugin_id_t`

7.2 Include

This section describes the `allinea_*.h` files.

- `allinea_metric_plugin_api.h`

  Header for the Linaro Forge sampler metric plugin API, includes all other API header files.

- `allinea_metric_plugin_errors.h`

  Functions for reporting errors encountered by a metric plugin library or specific metric.

- `allinea_metric_plugin_template.h`

  Header containing declarations for functions to be implemented by any Linaro MAP metric plugin library.

- `allinea_metric_plugin_types.h`

  Types and typedefs used by the Linaro MAP metric plugin API.

- `allinea_safe_malloc.h`

  Async signal safe memory management functions for use in metric plugins.

- `allinea_safe_syscalls.h`

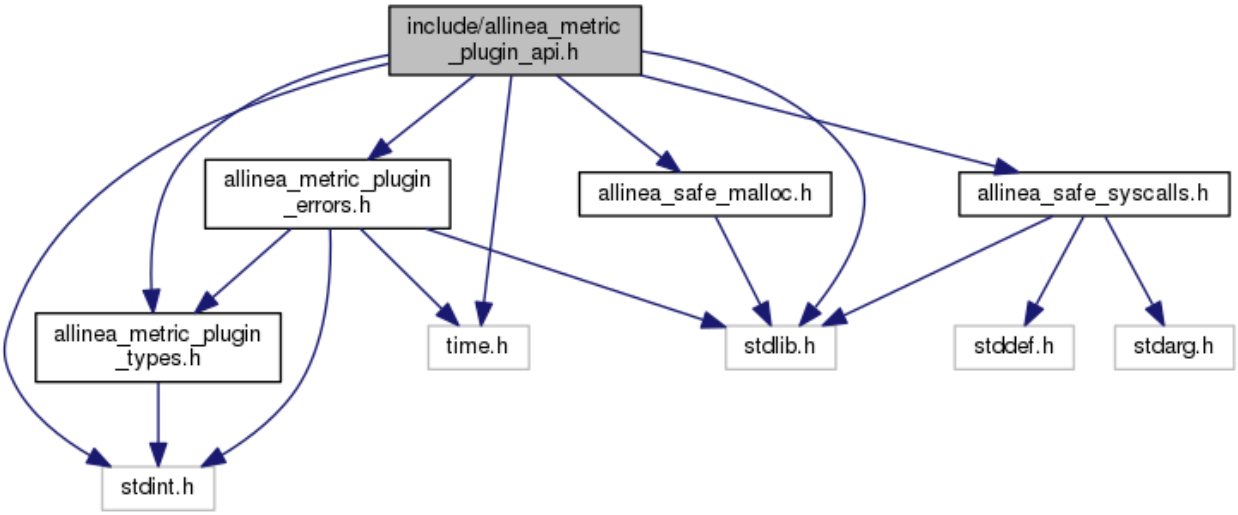
  Async signal safe I/O functions for use in metric plugins.

7.2.1 allinea_metric_plugin_api.h File Reference

This Header for the Linaro Forge sampler metric plugin API includes all other API header files.

```
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#include **allinea_metric_plugin_types.h**
#include **allinea_metric_plugin_errors.h**
#include **allinea_safe_malloc.h**
#include **allinea_safe_syscalls.h**
```

Include dependency graph for `allinea_metric_plugin_api.h`:



Source code

```
1
5 #ifndef ALLINEA_METRIC_PLUGIN_API_H
6 #define ALLINEA_METRIC_PLUGIN_API_H
7
8 #include <stdint.h>
9 #include <stdlib.h>
10 #include <time.h>
11
12 #include **allinea_metric_plugin_types.h**
13 #include **allinea_metric_plugin_errors.h**
14 #include **allinea_safe_malloc.h**
15 #include **allinea_safe_syscalls.h**
16
```

(continues on next page)

(continued from previous page)

```
17 #ifdef __cplusplus
18 extern **C** {
19 #endif
20
25
31 int allinea_get_logical_core_count(void);
32
34
40 int allinea_get_physical_core_count(void);
41
43
66 int allinea_read_config_file(const char *variable, const char *metricId, char *value, int length);
67
73 const char* allinea_get_custom_data(metric_id_t metricId);
74
76
77 #ifdef __cplusplus
78 }
79 #endif
80
81 #endif // ALLINEA_METRIC_PLUGIN_API_H
```

System info functions

Functions that provide information about the system or the enclosing profiler.

int allinea_get_logical_core_count() (void)
Returns the number of logical cores on this system.

int allinea_get_physical_core_count() (void)
Returns the number of physical cores on this system.

int allinea_read_config_file() (const char * variable, const char * metricId, char * value, int length)
Reads the configuration file to find the value of a variable.

const char * allinea_get_custom_data() (metric_id_t metricId)
Returns the **customData** attribute of the **source** element from the metric definition defined in the xml file.

See also

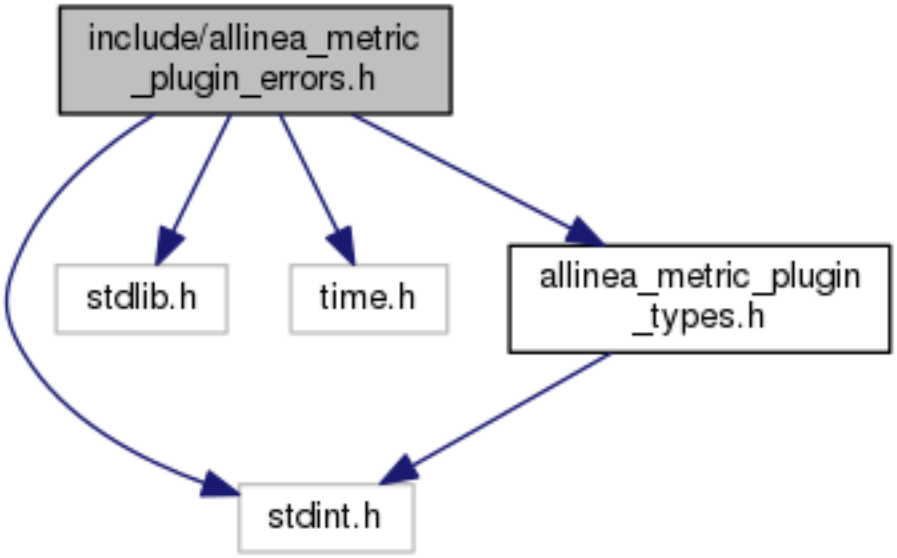
- *System info functions*
- *Standard utility functions*
- *metric_id_t*

7.2.2 allinea_metric_plugin_errors.h File Reference

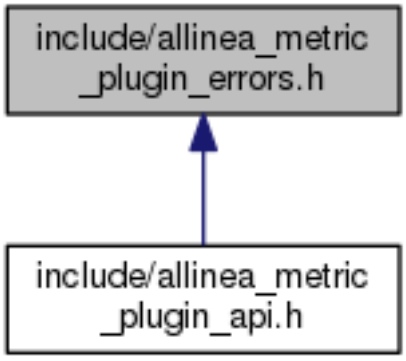
Functions for reporting errors encountered by a metric plugin library or specific metric.

```
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#include "allinea_metric_plugin_types.h
```

Include dependency graph for `allinea_metric_plugin_errors.h`:



This graph shows which files directly or indirectly include this file:



Source code

```
1
5 #ifndef ALLINEA_METRIC_PLUGIN_ERRORS_H
6 #define ALLINEA_METRIC_PLUGIN_ERRORS_H
7
8 #include <stdint.h>
9 #include <stdlib.h>
10 #include <time.h>
11
12 #include "allinea_metric_plugin_types.h"
13
14 #ifdef __cplusplus
15 extern "C" {
16 #endif
17
23
49 void allinea_set_plugin_error_message(plugin_id_t plugin_id, int error_code, const char *error_
 ↪message);
50
52
78 void allinea_set_plugin_error_messagef(plugin_id_t plugin_id, int error_code, const char *error_
 ↪message, ...);
79
81
98 void allinea_set_metric_error_message(metric_id_t metric_id, int error_code, const char *error_
 ↪message);
99
101
123 void allinea_set_metric_error_messagef(metric_id_t metric_id, int error_code, const char *error_
 ↪message, ...);
124
125
127
128 #ifdef __cplusplus
129 }
130 #endif
131
132 #endif // ALLINEA_METRIC_PLUGIN_ERRORS_H
```

Error reporting functions

Functions for reporting errors encountered by either a specific metric or an entire metric plugin library.

void allinea_set_plugin_error_message() (plugin_id_t plugin_id, int error_code, const char * error_message)
Reports an error that occurred in the plugin (group of metrics).

void allinea_set_plugin_error_messagef() (plugin_id_t plugin_id, int error_code, const char * error_message,...)
Reports an error occurred in the plugin (group of metrics).

void allinea_set_metric_error_message() (metric_id_t metric_id, int error_code, const char * error_message)
Reports an error occurred when reading a metric.

void allinea_set_metric_error_messagef() (metric_id_t metric_id, int error_code, const char * error_message,...)

Reports an error occurred when reading a metric.

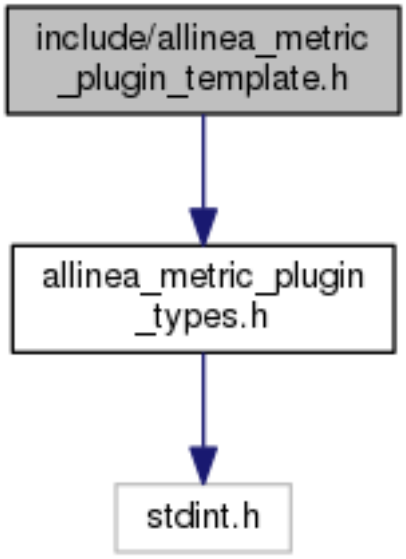
See also * *Error reporting functions* * *metric_id_t*

7.2.3 allinea_metric_plugin_template.h File Reference

Header containing declarations for functions that are implemented by any Linaro MAP metric plugin library.

`#include "allinea_metric_plugin_types.h"`

Include dependency graph for `allinea_metric_plugin_template.h`:



Source code

```
  1
  6 #ifndef ALLINEA_METRIC_PLUGIN_TEMPLATE_H
  7 #define ALLINEA_METRIC_PLUGIN_TEMPLATE_H
  8
  9 #include "allinea_metric_plugin_types.h"
 10
 11 #ifdef __cplusplus
 12 extern "C" {
 13 #endif
 14
 16
 36 int allinea_plugin_initialize(plugin_id_t plugin_id, void* data);
 37
 39
 58 int allinea_plugin_cleanup(plugin_id_t plugin_id, void* data);
 59
 61
113 int mymetric_getIntValue(metric_id_t id, struct timespec *currentSampleTime, uint64_t *outValue);
114
116
168 int mymetric_getDoubleValue(metric_id_t id, struct timespec *currentSampleTime, double *outValue);
```

(continues on next page)

(continued from previous page)

```
169
171
192 int start_profiling(plugin_id_t plugin_id);
193
195
216 int stop_profiling(plugin_id_t plugin_id);
217
218
219 #ifdef __cplusplus
220 }
221 #endif
222
223 #endif // ALLINEA_METRIC_PLUGIN_TEMPLATE_H
```

Functions

int allinea_plugin_cleanup (plugin_id_t plugin_id, void * data)
Cleans a metric plugin being unloaded.

int allinea_plugin_initialise (plugin_id_t plugin_id, void * data)
Initializes a metric plugin.

int mymetric_getDoubleValue (metric_id_t id, struct timespec * currentSampleTime, double * outValue)
Example of a floating-point metric *getter* function.

int mymetric_getIntValue (metric_id_t id, struct timespec * currentSampleTime, uint64_t * outValue)
Example of an integer metric *getter* function.

int start_profiling (plugin_id_t plugin_id)
Called when the Linaro Forge sampler is initialized.

int stop_profiling (plugin_id_t plugin_id)
Called after the Linaro Forge sampler stops sampling.

See also

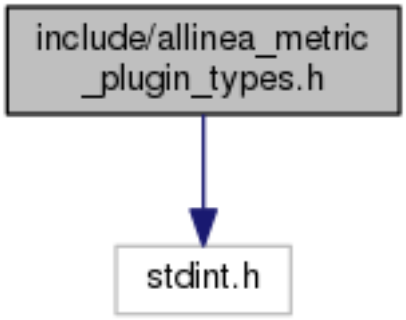
- *allinea_plugin_cleanup*
- *plugin_id_t*

7.2.4 allinea_metric_plugin_types.h File Reference

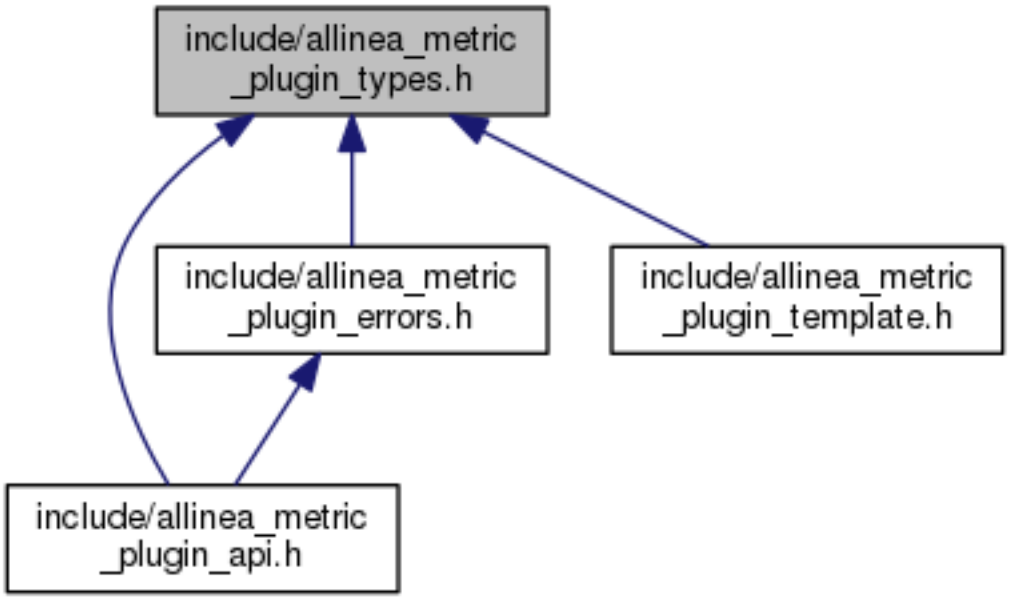
Types and typedefs used by the Linaro MAP metric plugin API.

`#include <stdint.h>`

Include dependency graph for `allinea_metric_plugin_types.h`:



This graph shows which files include this file directly or indirectly:



allinea_metric_plugin_types.h Source code

```
 1
 5 #ifndef ALLINEA_METRIC_PLUGIN_TYPES_H
 6 #define ALLINEA_METRIC_PLUGIN_TYPES_H
 7
 8 #include <stdint.h>
 9
10 #ifdef __cplusplus
11 extern "C" {
12 #endif
13
15 typedef uintptr_t plugin_id_t;
17 typedef uintptr_t metric_id_t;
18
19
20 #ifdef __cplusplus
21 }
22 #endif
```

(continues on next page)

(continued from previous page)

```
23
24 #endif // ALLINEA_METRIC_PLUGIN_TYPES_H
```

Typedefs

typedef uintptr_t metric_id_t
Opaque handle to a metric.

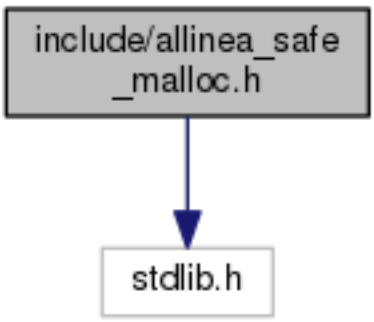
typedef uintptr_t plugin_id_t
Opaque handle to a metric plugin.

7.2.5 allinea_safe_malloc.h File Reference

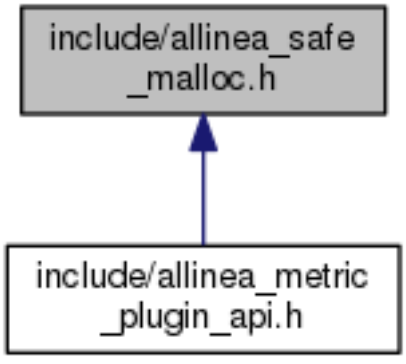
Async signal safe memory management functions for use in metric plugins.

`#include <stdlib.h>`

Include dependency graph for `allinea_safe_malloc.h`:



This graph shows which files directly or indirectly include this file:



Source code

```
1
5 #ifndef ALLINEA_SAFE_MALLOC_H
6 #define ALLINEA_SAFE_MALLOC_H
7
8 #include <stdlib.h>
9
10 #ifdef __cplusplus
11 extern "C" {
12 #endif
13
24
37 extern void* allinea_safe_malloc(size_t size);
38
40
49 extern void allinea_safe_free(void *ptr);
50
52
67 extern void* allinea_safe_calloc(size_t nmemb, size_t size);
68
70
89 extern void* allinea_safe_realloc(void *ptr, size_t size);
90
92
93 #ifdef __cplusplus
94 }
95 #endif
96
97 #endif
```

Memory management functions

Async signal safe replacements for memory management functions.

The metric library functions must be async signal safe and therefore the standard libc memory management functions (such as `malloc`, `free`, `new`, `delete`) cannot be used.

These memory management functions can safely be used by the metric plugin libraries even if they are called from inside a signal handler:

`void * allinea_safe_malloc (size_t size)`
: An async-signal-safe version of `malloc`.

`void allinea_safe_free (void * ptr)`
: An async-signal-safe version of `free`.

`void * allinea_safe_calloc (size_t nmemb, size_t size)`
: An async-signal-safe version of `calloc`.

`void * allinea_safe_realloc(void * ptr, size_t size)`
: An async-signal-safe version of `realloc`.

See also

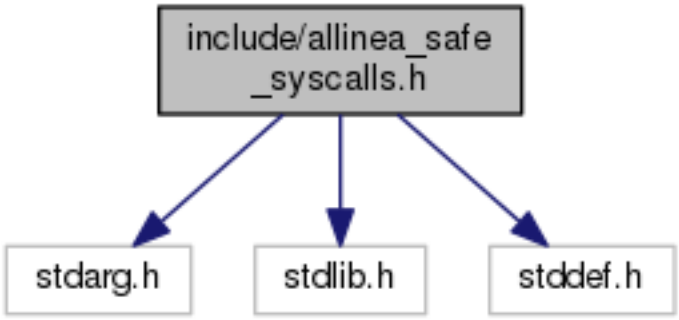
- *Memory management functions*
- *Metric plugin API*

7.2.6 allinea_safe_syscalls.h File Reference

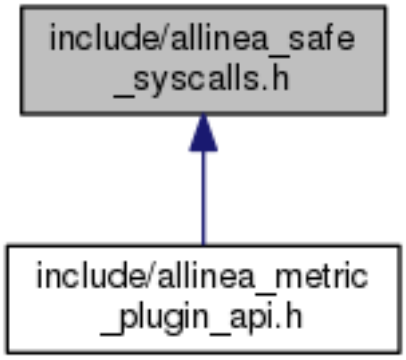
Async signal safe I/O functions for use in metric plugins.

- `#include <stdarg.h>`
- `#include <stdlib.h>`
- `#include <stddef.h>`

Include dependency graph for `allinea_safe_syscalls.h`:



This graph shows which files directly or indirectly include this file:



Source code

```
1
5 #ifndef ALLINEA_SAFE_SYSCALLS_H
6 #define ALLINEA_SAFE_SYSCALLS_H
7
8 #ifdef __cplusplus
9 extern "C" {
10 #endif
11
12 #include <stdarg.h>
13 #include <stdlib.h>
14 #include <stddef.h>
15
27
33 struct timespec allinea_get_current_time(void);
```

(continues on next page)

(continued from previous page)

```
 34
 35
 37
 44 extern int allinea_safe_close(int fd);
 45
 46
 48
 53 extern void allinea_safe_fprintf(int fd, const char *format, ...);
 54
 55
 57
 67 extern int allinea_safe_open (const char *file, int oflags, ...);
 68
 69
 71
 75 extern void allinea_safe_printf(const char *format, ...);
 76
 77
 79
 88 extern ssize_t allinea_safe_read(int fd, void *buf, size_t count);
 89
 90
 92
101 extern ssize_t allinea_safe_read_all(int fd, void *buf, size_t count);
102
103
105
115 extern ssize_t allinea_safe_read_all_with_alloc(int fd, void **buf, size_t *count);
116
117
132 extern ssize_t allinea_safe_read_line(int fd, void *buf, size_t count);
133
134
136
141 extern void allinea_safe_vfprintf(int fd, const char *format, va_list ap);
142
143
145
154 extern ssize_t allinea_safe_write(int fd, const void *buf, size_t count);
155
156
158
159 #ifdef __cplusplus
160 }
161 #endif
162
163 #endif
```

Standard utility functions

Replacements for common libc utility functions.

Most standard libc functions cannot be used because metric library functions need to be async signal safe. In addition, even basic syscalls (such as `read` and `write`) cannot be used without risking corruption of some other metrics the enclosing profiler might be tracking (for example, bytes read or bytes written).

These functions can be safely called inside signal handlers and accommodates I/O being done by the metric plugin without corrupting I/O metrics that are tracked by the enclosing profiler:

`struct timespec (void)`
: Gets the current time using the same clock as the enclosing profiler (async-signal-safe).

`int (int fd)`
: Closes the file descriptor `fd` previously opened by `allinea_safe_open` (async-signal-safe). See *Memory management functions*.

`void (int fd, const char * format,...)`
: An async-signal-safe version of `fprintf`.

`int (const char * file, int oflags,...)`
: Opens the given file for reading or writing (async-signal-safe).

`void (const char * format,...)`
: An async-signal-safe replacement for `printf`.

`ssize_t (int fd, void * buf, size_t count)`
: Reads up to *count* bytes from `buf` to `fd` (async-signal-safe)

`ssize_t (int fd, void * buf, size_t count)`
: Reads the entire contents of `fd` into `buf` (async-signal-safe).

`ssize_t (int fd, void * buf, size_t * count)`
: Reads the entire contents of `fd` into `buf` (async-signal-safe).

`ssize_t (int fd, void * buf, size_t count)`
: Reads a line from `fd` into `buf` (async-signal-safe).

`void (int fd, const char * format, va_list ap)`
: An async-signal-safe version of `vfprintf`.

`ssize_t (int fd, const void * buf, size_t count)`
: Writes up to **count** bytes from `buf` to `fd` (async-signal-safe).

See also

- *Memory management functions*
- *Standard utility functions*
- *Metric plugin API*

CHAPTER

EIGHT

EXAMPLES

This section describes the provided examples.

8.1 backfill1.c

An example of a backfilled custom metric.

This category of metric allows data collected externally from the Linaro Forge sampler (for example, hardware power monitoring or I/O logs), to display alongside metrics which are collected by the Linaro Forge sampler.

```
#include "allinea_metric_plugin_api.h"
int allinea_plugin_initialise(plugin_id_t plugin_id, void *unused)
{
    return 0;
}

int allinea_plugin_cleanup(plugin_id_t plugin_id, void *unused)
{
    return 0;
}

int start_profiling(plugin_id_t plugin_id)
{
    return 0;
}

int stop_profiling(plugin_id_t plugin_id)
{
    return 0;
}

int backfilled_metric(metric_id_t metric_id, struct timespec *in_out_sample_time, uint64_t *out_value)
{
    // Back fill with value of 5 for all samples.
    *out_value = 5;
    return 0;
}
```

8.2 backfill1.xml

An example of a definition for a backfilled metric.

This example corresponds to the source in *backfill1.c*. The key difference when compared with the definition for a metric sampled at runtime (for example, *custom1.xml*) is that the `backfill` attribute is set to `true`.

```
<metricdefinitions version="1">
    <metric id="com.allinea.metrics.backfill1.events">
        <units>/s</units>
        <dataType>uint64_t</dataType>
        <domain>time</domain>
        <backfill>true</backfill>
        <source ref="com.allinea.metrics.backfill_src"
                functionName="backfilled_metric"/>
        <display>
            <displayName>Events</displayName>
            <description>Total number of events</description>
            <type>events</type>
            <colour>red</colour>
         </display>
    </metric>
    <metricGroup id="Backfill1">
        <displayName>Backfill1</displayName>
        <description>Number of events</description>
        <metric ref="com.allinea.metrics.backfill1.events"/>
    </metricGroup>
    <source id="com.allinea.metrics.backfill_src">
        <sharedLibrary>libbackfill1.so</sharedLibrary>
        <functions>
            <start>start_profiling</start>
            <stop>stop_profiling</stop>
        </functions>
    </source>
</metricdefinitions>
```

8.3 custom1.c

An example metric library using the Linaro MAP Metric Plugin API.

This implements the functions as defined in *Metric Plugin Template* and makes calls to the *Metric plugin API*. This plugin provides a custom metric showing the number of interrupts handled by the system, as obtained from `/proc/interrupts`.

Compile the plugin using this command:

```
gcc -fPIC -I/path/to/forge/metrics/include -shared -o libcustom1.so custom1.c
```

For the corresponding definition file to enable the `libcustom1.so` metric library to be used by compatible profilers, see *custom1.xml*.

```
/* The following functions are assumed to be async-signal-safe, although not
* required by POSIX:
*
* strchr strstr strtoull
```

(continues on next page)

(continued from previous page)

```
*/

#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "allinea_metric_plugin_api.h"

#define PROC_STAT "/proc/stat"

#define ERROR_NO_PROC_STAT                        1000

#define BUFSIZE     256
#define OVERLAP    64

#ifndef min
#define min(x, y) ( ((x) < (y)) ? (x) : (y) )
#endif

static uint64_t previous = 0;
static int have_previous = 0;

int allinea_plugin_initialise(plugin_id_t plugin_id, void *unused)
{
    // Check that /proc/interrupts exists.
    if (access(PROC_STAT, F_OK) != 0) {
        if (errno == ENOENT)
            allinea_set_plugin_error_messagef(plugin_id, ERROR_NO_PROC_STAT,
                "Not supported (no /proc/interrupts)");
        else
            allinea_set_plugin_error_messagef(plugin_id, errno,
                "Error accessing %s: %s", PROC_STAT, strerror(errno));
        return -1;
    }
}

int allinea_plugin_cleanup(plugin_id_t plugin_id, void *unused)
{
}

int sample_interrupts(metric_id_t metric_id, struct timespec *in_out_sample_time, uint64_t *out_value)
{
    // Main buffer. Add an extra byte for the '\0' we add below.
    char buf[BUFSIZE + 1];
    *in_out_sample_time = allinea_get_current_time();
    // We must use the allinea_safe variants of open / read / write / close so
    // that we are not included in the I/O accounting of the Forge sampler.
    const int fd = allinea_safe_open(PROC_STAT, O_RDONLY);
    if (fd == -1) {
        allinea_set_metric_error_messagef(metric_id, errno,
            "Error opening %s: %d", PROC_STAT, strerror(errno));
        return -1;
    }
    for (;;) {
        const ssize_t bytes_read = allinea_safe_read_line(fd, buf, BUFSIZE);
        if (bytes_read == -1) {
            // read failed
```

(continues on next page)

(continued from previous page)

```
            allinea_set_metric_error_messagef(metric_id, errno,
                "Error opening %s: %d", PROC_STAT, strerror(errno));
            break;
        }
        if (bytes_read == 0) {
            // end of file
            break;
        }
        if (strncmp(buf, "intr ", 5)==0) { // Check if this is the interrupts line.
            // The format of the line is:
            // intr <total> <intr 1 count> <intr 2 count> ...
            // Check we have the total by looking for the space after it.
            const char *total = buf + /* strlen("intr ") */ 5;
            char *space = strchr(total, ' ');
            if (space) {
                uint64_t current;
                // NUL-terminate the total.
                *space = '\0';
                // total now points to the NUL-terminated total. Convert it to
                // an integer.
                current = strtoull(total, NULL, 10);
                if (have_previous)
                    *out_value = current - previous;
                previous = current;
                have_previous = 1;
                break;
            }
        }
    }
    allinea_safe_close(fd);
}
```

8.4 custom1.xml

An example metric definition file.

This example is detailed in *Metric Definition File*. This corresponds to the example metric library *custom1.c*.

```
1 <!-- version is the file format version -->
2 <metricdefinitions version="1">
3
4     <!-- id is the internal name for this metric, as used in the .map XML -->
5     <metric id="com.allinea.metrics.custom1.interrupts">
6         <!-- Specify whether this metric is always, default_yes, default_no, or never enabled -->
7         <enabled>default_yes</enabled>
8         <!-- The units this metric is measured in. -->
9         <units>/s</units>
10         <!-- Data type used to store the sample values, uint64_t or double -->
11         <dataType>uint64_t</dataType>
12         <!-- The domain the metric is to be sampled in, only time is supported. -->
13         <domain>time</domain>
14
15         <!-- Example source
16              Specifies the source of data for this metric, i.e. a function in a
```

(continues on next page)

(continued from previous page)

```
17              shared library.
18
19              The function signature depends on the dataType:
20              - uint64_t:  int function(metric_id_t metricId,
21                                        struct timespec* inCurrentSampleTime,
22                                        uint64_t *outValue);
23              - double:    int function(metric_id_t metricId,
24                                        struct timespec* inCurrentSampleTime,
25                                        double   *outValue);
26
27              If the result is undefined for some reason the function may return
28              the special sentinel value ~0 (unsigned integers) or Nan (floating point)
29
30              Return value is 0 if success, -1 if failure (and set errno)
31
32              If divideBySampleTime is true then the values returned by outValue
33              will be divided by the sample interval to get the final value.  -->
34          <source ref="com.allinea.metrics.custom1_src"
35                  functionName="sample_interrupts"
36                  divideBySampleTime="true"/>
37
38          <!-- Display attributes used by the GUI -->
39          <display>
40              <!-- Display name for the metric as used in the GUI -->
41              <displayName>Interrupts</displayName>
42
43              <!-- Brief description of the metric.. -->
44              <description>Total number of system interrupts taken</description>
45
46              <!-- The type of metric, used by the GUI to group metrics -->
47              <type>interrupts</type>
48
49              <!-- The colour to use for the metric graphs for this metric -->
50              <colour>green</colour>
51          </display>
52
53      </metric>
54
55      <!-- Metric group for interrupt metrics, used in the GUI -->
56      <metricGroup id="Custom1">
57          <!-- Display name for the group as use din the GUI -->
58          <displayName>Custom1</displayName>
59
60          <!-- Brief description of the group -->
61          <description>Interrupt metrics</description>
62
63          <!-- References to all the metrics included in the group -->
64          <metric ref="com.allinea.metrics.custom1.interrupts"/>
65      </metricGroup>
66
67      <!-- Definition of the example source (metric plugin) used for the custom metric -->
68      <source id="com.allinea.metrics.custom1_src">
69          <!-- File name of the sample metric plugin shared library -->
70          <sharedLibrary>libcustom1.so</sharedLibrary>
71      </source>
72
73  </metricdefinitions>
```

8.5 report.xml

An example partial report definition file.

This example is detailed in *Linaro Performance Reports Integration*. This informs Linaro Performance Reports of the custom metrics implemented in *custom1.c*.

```
1 <partialReport name="InterruptsReport"
2               xmlns="http://www.allinea.com/2016/AllineaReports">
3   <reportMetrics>
4     <!-- multiple <reportMetric> elements can be defined -->
5     <!-- source attribute must be set to "metric" -->
6     <reportMetric id="interrupts.mean"
7                   displayName="Mean interrupts"
8                   units="/s"
9                   source="metric"
10                   colour="hsl(25, 70, 71)">
11       <sourceDetails metricRef="com.allinea.metrics.custom1.interrupts" sampleValue="mean"␣
 ↪aggregation="mean"/>
12     </reportMetric>
13     <reportMetric id="interrupts.peak"
14                   displayName="Peak interrupts"
15                   units="/s"
16                   source="metric"
17                   colour="hsl(19, 70, 71)">
18       <sourceDetails metricRef="com.allinea.metrics.custom1.interrupts" sampleValue="max"␣
 ↪aggregation="max"/>
19     </reportMetric>
20   </reportMetrics>
21   <subsections>
22     <!-- multiple <subsection> elements can be defined -->
23     <subsection id="interrupt_metrics"
24                 heading="Interrupts"
25                 colour="hsl(21, 70, 71)">
26       <text>The number of CPU interrupts raised per second across all ranks</text>
27       <!-- multiple <entry> elements can be defined -->
28       <entry reportMetric="interrupts.mean" group="InterruptsGroup"/>
29       <entry reportMetric="interrupts.peak" group="InterruptsGroup"/>
30     </subsection>
31   </subsections>
32 </partialReport>
```

CHAPTER

NINE

PROPRIETARY NOTICE

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Linaro. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated**.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED AS IS. LINARO PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Linaro makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL LINARO BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF LINARO HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word partner in reference to Linaro's customers is not intended to create or refer to any partnership relationship with any other company. Linaro may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Linaro, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Linaro corporate logo and words marked with or are registered trademarks or trademarks of Linaro Limited in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Linaro's trademark usage guidelines at https://www.linaro.org/legal#trademark-usage.

Copyright © 2023-2026 Linaro Limited. All rights reserved

Copyright ©2017-2023 Arm Limited (or its affiliates). All rights reserved.

Copyright ©2002-2017 Allinea Software Limited.

Linaro Limited. Company 07180318 registered in England. Harston Mill, Harston, Cambridge, CB22 7GG, UK.

Confidentiality status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Linaro and the party that Linaro delivered this document to.

Product status

The information in this document is Final, that is for a developed product.

Release history

https://www.linaroforge.com/release-history

Web address

https://www.linaro.org/

https://www.linaroforge.com/