Arm MAP Metric Plugin Interface

Generated by Doxygen 1.8.11

Contents

1 Summary

1.1 Introduction

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

1.2 Documentation

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

- The metric plugin template
- The metric plugin API
- The metric plugin definition file
- Arm 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 sampler is initialized and when sampling ends.

The metric plugin API documents the functions that may 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.

Arm 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 Arm Performance Reports.

1.3 Advice to metric authors

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

1. Speed
2. Async-signal safety

Arm 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* should return values as fast as possible, and long-running operations should be avoided unless in the allinea_plugin_initialize() or allinea_plugin_cleanup() functions.

Arm MAP does its sampling from inside a signal handler. This signal may 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.

1.4 Advice to profiler authors

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

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

1.5 Static linking

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

1.5.1 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 (i.e. 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.

2 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 should process & 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 below:

2.1 <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, this should not contain any whitespace and should be chosen to minimize the risk of clashing with an existing metric name. To avoid this, derive your metric ids from your website domain name and the name of 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 via the command line. A metrics source library will not be loaded if all metrics which it defines have been disabled.

units:

The units this metric is measured in. The profiler may 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 may be specified but the profiler, may 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`, the library only is enabled in one process of the node, and this process will be 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` is 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` is 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 should divide 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 will also be multiplied by 100.0 to give a percentage value in the range 0..100 (in this situation the function specified by `functionName` should return a time-in-seconds value).
- `customData` is a 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 should be presented:

- `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 colour to use when displaying the metric (for example, the colour of any graphs generated using this metric). This field is optional - the profiler may choose to use a colour based on the `type` field instead of the colour code specified here. Colour values may 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 may 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"/>
```

2.2 <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 may 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.

2.3 <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 will be referenced as in this .xml file. This id is used within the `source` fields of <metric> elements.

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 may 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 sampler is initialised. This function is optional but must have the same signature as start_profiling().

stop:

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

3 Arm Performance Reports Integration

3.1 Introduction

To integrate one or more metric plugins into Arm Performance Reports, provide a small 'partial report' file that describes the additions to be made to a Performance Report. This report should display '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).

3.2 Default definition file location

This custom partial report content can be defined in one or multiple custom report definition files, that have to 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: The default configuration folders sub-folder structure must be the same in order to use this feature (for example, *$ALLINEA_CONFIG_DIR/perf-report/reports/*).

3.3 Custom definition file location

The location of the custom partial report definition file(s) can be overriden 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 will be loaded in ascending alphabetical order.

3.4 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>
    <!-- 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>
```

3.4.1 <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.

3.4.2 <reportMetrics>

Container for one or more `<reportMetric>` elements.

3.4.3 <reportMetric>

```
<reportMetric id="com.allinea.metrics.sample.average" displayName="Average of custom interrupt metric"
      tooltip="Metric tooltip." units="/s" colour="hsl(19, 70, 71)"
            source="metric">
  <sourceDetails metricRef="com.allinea.metrics.sample.interrupts" sampleValue="mean" aggregation="mean"/>
</reportMetric>
```

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

id:

A unique identifier for this metric. This will be used to reference this metric in <entry> element(s). Ids with dots in their value should 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 will appear 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 will appear 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 should scale custom metrics and select custom units accordingly.

colour (optional):

The colour to be used for this metric when it is named in the report. Colour is also used for comparison bars showing the relative value of this metric. See Colour codes.

source:

Source type of this metric, must be set to `metric`.

3.4.4 <sourceDetails>

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

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

metricRef

Reference to an existing metric, can be either an Arm 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*).

3.4.5 <subsections>

Container for one or more <subsection> elements.

3.4.6 <subsection>

```
<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>
```

Describes the layout of the data to be displayed.

id:

A unique identifier for this subsection.

heading:

The display name for this subsection.

colour (optional):

The colour to use for the heading text. See Colour codes.

3.4.7 <text>

```
<text>Section one:</text>
```

Short static description of this subsection.

3.4.8 <entry>

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

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

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.

3.5 Colour codes

Colours may 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)` (each of X, Y, and Z is a decimal value 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 colours defined in the list of *SVG colour keyword* names provided by the World Wide Web Consortium: `https://www.w3.org/TR/SVG/types.html#ColorKeywords`

3.6 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`) - it cannot contain symbols (except '.', '_' and '-'). Note: IDs cannot begin with '.' or '_'.

3.7 HTML Markup

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

4 Quick Start

Following the instructions in this section is the quickest way to get started using custom metrics with Arm MAP and Arm Performance Reports.

1. Open a terminal in the `/custom/examples/` directory. In this directory you will find:
   - a Makefile for building the custom metrics shared library.
   - the source for the example custom metric (custom1.c).
   - report.xml, which explains to Arm Performance Reports how to access the custom metric.
   - custom1.xml, which provides metadata about this metric to Arm MAP.
2. If a custom configuration directory for the Arm HPC tools 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 Arm MAP. To display the custom metric upon completion of the run, use the Metrics menu (Metrics -> Preset: Custom1). An example of how this looks is given in the figure below. In addition, the .html and the .txt report files generated by Arm Performance Reports will have an additional section containing the custom metric data.

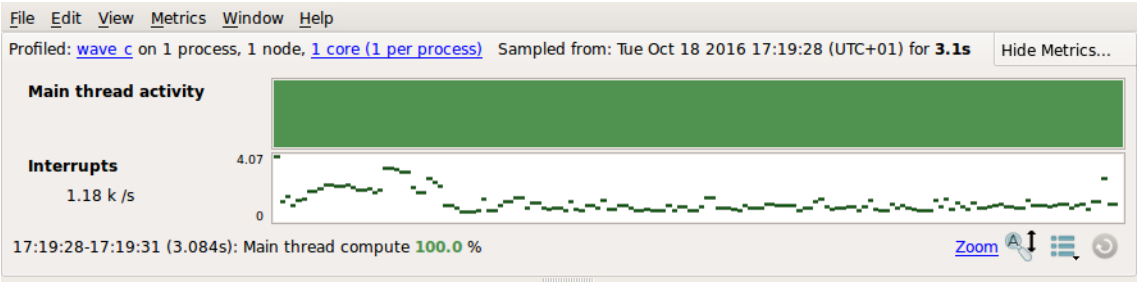

Figure 1 Custom Interrupt Metric

4.1 Custom Metric Development

The development of custom metrics for use with Arm 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 details the meta information in custom1.xml that Arm 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, information on exposing custom metrics in Arm Performance Reports is provided in Arm Performance Reports Integration.

5 Module Index

5.1 Modules

Here is a list of all modules:

6 File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

7 Module Documentation

7.1 Metric Plugin API

The API functions available for use by a metric plugin library.

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)
  It returns the "customData" attribute of the "source" element from the metric definition defined in the xml file.

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.

Memory management functions

Async signal safe replacements for memory management functions.

Since 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`.
- 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`.

Standard Utility Functions

Replacements for common libc utility functions.

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

  Closes the file descriptor fd previously opened by allinea_safe_open (async-signal-safe).
- void allinea_safe_fprintf (int fd, const char *format,...)

  An async-signal-safe version of `fprintf`.
- int allinea_safe_open (const char *file, int oflags,...)

  Opens the given file for reading or writing (async-signal-safe).
- void allinea_safe_printf (const char *format,...)

  An async-signal-safe replacement for `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_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_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_line (int fd, void *buf, size_t count)

  Reads a line from fd into buf (async-signal-safe).
- void allinea_safe_vfprintf (int fd, const char *format, va_list ap)

  An async-signal-safe version of `vfprintf`.
- 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).

7.1.1 Detailed Description

The API functions available for use by a metric plugin library.

7.1.2 Function Documentation

7.1.2.1 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 (i.e. `CLOCK_MONOTONIC`).

Returns

The current time

Examples:

custom1.c.

7.1.2.2 const char* allinea_get_custom_data (metric_id_t *metricId*)

It returns the "customData" attribute of the "source" element from the metric definition defined in the xml file.

Parameters

metricId	metric id

Returns

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

7.1.2.3 int allinea_get_logical_core_count (void)

Returns 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

7.1.2.4 int allinea_get_physical_core_count (void)

Returns the number of physical cores on this system.

This count does *not* include the *effective* cores 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

7.1.2.5 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.

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 sampler (such as in allinea_plugin_initialise and similar functions) as it is not async signal safe.

Parameters

in	*variable*	The name of the configuration variable.
in	*metric↩ Id*	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.

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

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

Allocates `size * nmemb` bytes and zero-initialises the memory.

To be used 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 should be released by a call to allinea_safe_free(). Do not use libc `free` to free memory allocated by this function.

Parameters

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.

7.1.2.7 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 will not be included in the enclosing profiler's I/O accounting.

Parameters

fd	The file descriptor to close.

Returns

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

Examples:

custom1.c.

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

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

Parameters

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.

7.1.2.9 void allinea_safe_free (void * *ptr*)

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

Frees a memory region previously allocated with allinea_safe_malloc.

To be used instead of the libc `free`. Do not use this function to deallocate memory blocks previously allocated by the libc `malloc`.

Parameters

`in,out`	*ptr*	A pointer to the start of the memory region to free. This should have been previously allocated with allinea_safe_malloc(), allinea_safe_realloc(), or allinea_safe_calloc().

7.1.2.10 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

`in`	*size*	The number of bytes of memory to allocate.

Returns

a pointer to the start of the allocated memory region.

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

Opens the given *file* for reading or writing (async-signal-safe).

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

Parameters

file	The name of the file to open (may be an absolute or relative path)
oflags	Flags specifying how the file should be opened. Accepts all the flags that may be given to the libc `open` function i.e. `O_RDONLY`, `O_WRONLY`, or `O_RDWR`.

Returns

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

Examples:

custom1.c.

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

An async-signal-safe replacement for `printf`.

Parameters

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

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

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

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

Parameters

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.

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

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

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

Parameters

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 will be written to *buf*.

Returns

If successful, the number of bytes read, else -1 and *errno* is set.

7.1.2.15 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).

When used in conjunction with allinea_safe_open() and allinea_safe_close(), the read bytes will be excluded from the enclosing profiler's I/O accounting. Sufficient space for the file contents plus a terminating NUL is allocated and should be freed, using allinea_safe_free, when no longer required.

Parameters

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

Returns

If successful the number of bytes read, else -1 and *errno* is set.

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

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

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

Lines longer than *count* will be truncated.

Parameters

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 will be written to *buf*.

Returns

If successful, the number of bytes read, else -1 and *errno* is set.

Examples:

custom1.c.

7.1.2.17 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`.

To be used 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 should be allocated by a call to allinea_safe_malloc(), allinea_safe_calloc() or allinea_safe_realloc().

Memory allocated by this function should be released by a call to allinea_safe_free(). Do not use libc `free` to free memory allocated by this function.

Parameters

`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

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

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

Parameters

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

7.1.2.19 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).

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

Parameters

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.

7.1.2.20 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.

Parameters

metric_id	The id identifying the metric that has encountered an error. The appropriate value will have been passed in as an argument to the metric *getter* call.
error_code	An error code that can be used to distinguish between the possible errors that may have occurred. The exact value is up to the plugin author but each error condition should have its own and unique error code. In the case of a failing libc function the libc `errno` (from `<errno.h>`) may be appropriate, but a plugin-author-specified constant could also be used. The meaning of the possible error codes should be documented for the benefit of users of your plugin.
error_message	A text string describing the error in a human-readable form. In the case of a failing libc function the value `strerror(errno)` may be appropriate, but a plugin-author-specified message could also be used.

7.1.2.21 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.

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

Parameters

metric_id	The id identifying the metric that has encountered an error. The appropriate value will have been passed in as an argument to the metric *getter* call.
error_code	An error code that can be used to distinguish between the possible errors that may have occurred. The exact value is up to the plugin author but each error condition should have its own and unique error code. In the case of a failing libc function the libc `errno` (from `<errno.h>`) may be appropriate, but a plugin-author-specified constant could also be used. The meaning of the possible error codes should be documented for the benefit of users of your plugin.

Parameters

error_message	A text string describing the error in a human-readable form. In the case of a failing libc function the value `strerror(errno)` may be appropriate, but a plugin-author-specified message could also be used. This may include printf-style substitution characters.
...	Zero or more values to be substituted into the *error_message* string.

Examples:

custom1.c.

7.1.2.22 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).

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

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

Parameters

plugin_id	The id identifying the plugin that has encountered an error. The appropriate value will have been passed in as an argument to the allinea_plugin_initialize() call.
error_code	An error code that can be used to distinguish between the possible errors that may have occurred. The exact value is up to the plugin author but each error condition should have its own and unique error code. In the case of a failing libc function the libc `errno` (from `<errno.h>`) may be appropriate, but a plugin-author-specified constant could also be used. The meaning of the possible error codes should be documented for the benefit of users of your plugin.
error_message	A text string describing the error in a human-readable form. In the case of a failing libc function the value `strerror(errno)` may be appropriate, but a plugin-author-specified message could also be used.

7.1.2.23 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).

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

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

Parameters

plugin_id	The id identifying the plugin that has encountered an error. The appropriate value will have been passed in as an argument to the allinea_plugin_initialize() call.
error_code	An error code that can be used to distinguish between the possible errors that may have occurred. The exact value is up to the plugin author but each error condition should have its own and unique error code. In the case of a failing libc function the libc `errno` (from `<errno.h>`) may be appropriate, but a plugin-author-specified constant could also be used. The meaning of the possible error codes should be documented for the benefit of users of your plugin.

Parameters

error_message	A text string describing the error in a human-readable form. In the case of a failing libc function the value `strerror(errno)` may be appropriate, but a plugin-author-specified message could also be used. This may include printf-style substitution characters.
...	Zero or more values to be substituted into the *error_message* string in the same manner as printf.

Examples:

custom1.c.

7.2 Metric Plugin Template

The functions that must be implemented by every metric plugin library.

Functions

- int allinea_plugin_cleanup (plugin_id_t plugin_id, void *data)
  Cleans a metric plugin being unloaded.
- int allinea_plugin_initialize (plugin_id_t plugin_id, void *data)
  Initialises 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 sampler is initialised.
- int stop_profiling (plugin_id_t plugin_id)
  Called after the sampler stops sampling.

7.2.1 Detailed Description

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_initialize() 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 may be called by this metric library.

See Metric Definition File for information on the format of the definition file that will inform profilers what metrics the metric plugin library may provide.

7.2.2 Function Documentation

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

Cleans a metric plugin being unloaded.

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 will be called after metric data has been extracted and transferred to the frontend. Therefore, you may not see plugin error messages set by allinea_set_plugin_error_message() or allinea_set_plugin_error_messagef().

Parameters

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, will always be NULL

Returns

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

Examples:

backfill1.c, and custom1.c.

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

Initialises a metric plugin.

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 (e.g. the required information is not available on this machine) then it should call allinea_set_plugin_error_message() or allinea_set_plugin_error_messagef() to explain the situation then return -1.

Parameters

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, will always be NULL

Returns

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

Examples:

backfill1.c, and custom1.c.

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

Example of a floating-point metric *getter* function.

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 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 i.e. 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 sampler is now requesting a data point for.

This parameter is additionally an out parameter and may 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 current↩SampleTime from any other source is undefined. In the case of a backfilled metric, currentSampleTime does not function as an out parameter and will result in an error if it is used as such. It is safe to assume that this pointer is not NULL.

Parameters

`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 should call allinea_set_metric_error_message() before returning.

Warning

This function may 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 behaviour.

Note

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

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

Example of an integer metric *getter* function.

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 :

```
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 i.e. 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 sampler is now requesting a data point for.

This parameter is additionally an out parameter and may 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 current↩SampleTime from any other source is undefined. In the case of a backfilled metric, currentSampleTime does not function as an out parameter and will result in an error if it is used as such. It is safe to assume that this pointer is not NULL.

Parameters

`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 should call allinea_set_metric_error_message() before returning.

Warning

This function may 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 behaviour.

Note

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

7.2.2.5 int start_profiling (plugin_id_t *plugin_id*)

Called when the sampler is initialised.

An example of a function which is called when the sampler is initialised. 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

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 should be supplied using allinea_set_plugin_error_message() or allinea_set_plugin_error_messagef() before returning.

Examples:

backfill1.c.

7.2.2.6 int stop_profiling (plugin_id_t *plugin_id*)

Called after the sampler stops sampling.

An example of a function which is called when the 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 may be called from a signal handler so must be async-signal-safe

Parameters

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 should be supplied using allinea_set_plugin_error_message() or allinea_set_plugin_error_messagef() before returning.

Examples:

backfill1.c.

8 File Documentation

8.1 include/allinea_metric_plugin_api.h File Reference

Header for the Arm MAP 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:

8.2 include/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:

Functions

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.

8.2.1 Detailed Description

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

8.3 include/allinea_metric_plugin_template.h File Reference

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

```
#include "allinea_metric_plugin_types.h"
```

Include dependency graph for 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_initialize (plugin_id_t plugin_id, void *data)

  Initialises 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 sampler is initialised.
- int stop_profiling (plugin_id_t plugin_id)

  Called after the sampler stops sampling.

8.3.1 Detailed Description

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

8.4 include/allinea_metric_plugin_types.h File Reference

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

`#include <stdint.h>`
Include dependency graph for allinea_metric_plugin_types.h: This graph shows which files directly or indirectly include this file:

Typedefs

- typedef uintptr_t metric_id_t

  Opaque handle to a metric.
- typedef uintptr_t plugin_id_t

  Opaque handle to a metric plugin.

8.4.1 Detailed Description

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

8.5 include/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:

Functions

Memory management functions

Async signal safe replacements for memory management functions.

Since 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`.
- 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`.

8.5.1 Detailed Description

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

8.6 include/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:

Functions

Standard Utility Functions

Replacements for common libc utility functions.

Since metric library functions need to be async signal safe most standard libc functions cannot be used. In addition, even basic syscalls (such as `read` and `write`) cannot be used without risking corruption of some other metrics the enclosing profiler may be tracking (for example, bytes read or bytes written). The following functions can be safely called inside signal handlers and will accommodate I/O being done by the metric plugin without corrupting I/O metrics being 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).
- int allinea_safe_close (int fd)
  Closes the file descriptor fd previously opened by allinea_safe_open (async-signal-safe).
- void allinea_safe_fprintf (int fd, const char *format,...)
  An async-signal-safe version of `fprintf`.
- int allinea_safe_open (const char *file, int oflags,...)
  Opens the given file for reading or writing (async-signal-safe).
- void allinea_safe_printf (const char *format,...)
  An async-signal-safe replacement for `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_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_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_line (int fd, void ∗buf, size_t count)

  Reads a line from fd into buf (async-signal-safe).
- void allinea_safe_vfprintf (int fd, const char ∗format, va_list ap)

  An async-signal-safe version of `vfprintf`.
- 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).

8.6.1 Detailed Description

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

9 Example Documentation

9.1 backfill1.c

An example of a backfilled custom metric. This category of metric allows data, which has been collected externally to the sampler (for example, hardware power monitoring or I/O logs), to be displayed alongside metrics which are collected by the sampler.

```
#include "allinea_metric_plugin_api.h"

int allinea_plugin_initialize(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;
}
```

9.2 backfill1.xml

An example of a definition for a backfilled metric which 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`.

```
1 <metricdefinitions version="1">
2     <metric id="com.allinea.metrics.backfill1.events">
3         <units>/s</units>
4         <dataType>uint64_t</dataType>
5         <domain>time</domain>
6         <backfill>true</backfill>
7         <source ref="com.allinea.metrics.backfill_src"
8                 functionName="backfilled_metric"/>
9         <display>
10             <displayName>Events</displayName>
11             <description>Total number of events</description>
12             <type>events</type>
13             <colour>red</colour>
14         </display>
15     </metric>
16     <metricGroup id="Backfill1">
17         <displayName>Backfill1</displayName>
18         <description>Number of events</description>
19         <metric ref="com.allinea.metrics.backfill1.events"/>
20     </metricGroup>
21     <source id="com.allinea.metrics.backfill_src">
22         <sharedLibrary>libbackfill1.so</sharedLibrary>
23         <functions>
24             <start>start_profiling</start>
25             <stop>stop_profiling</stop>
26         </functions>
27     </source>
28 </metricdefinitions>
```

9.3 custom1.c

An example metric library using the Arm 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`.

It can be compiled using the command:

```
gcc -fPIC -I/path/to/arm/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
 */

#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_initialize(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 Arm MAP 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
            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);
}
```

9.4 custom1.xml

An example metric definition file, as detailed in Metric Definition File. This corresponds to the example metric library custom1.c.

```
<!-- version is the file format version -->
<metricdefinitions version="1">

    <!-- id is the internal name for this metric, as used in the .map XML -->
    <metric id="com.allinea.metrics.custom1.interrupts">
        <!-- Specify whether this metric is always, default_yes, default_no, or never enabled -->
        <enabled>default_yes</enabled>
        <!-- The units this metric is measured in. -->
        <units>/s</units>
         <!-- Data type used to store the sample values, uint64_t or double -->
         <dataType>uint64_t</dataType>
         <!-- The domain the metric is to be sampled in, only time is supported. -->
         <domain>time</domain>

         <!-- Example source
              Specifies the source of data for this metric, i.e. a function in a
              shared library.

              The function signature depends on the dataType:
              - uint64_t:  int function(metric_id_t metricId,
                                        struct timespec* inCurrentSampleTime,
                                        uint64_t *outValue);
              - double:    int function(metric_id_t metricId,
                                        struct timespec* inCurrentSampleTime,
                                        double   *outValue);

             If the result is undefined for some reason the function may return
             the special sentinel value ~0 (unsigned integers) or Nan (floating point)

             Return value is 0 if success, -1 if failure (and set errno)

             If divideBySampleTime is true then the values returned by outValue
             will be divided by the sample interval to get the final value.  -->
         <source ref="com.allinea.metrics.custom1_src"
                 functionName="sample_interrupts"
                 divideBySampleTime="true"/>

         <!-- Display attributes used by the GUI -->
         <display>
             <!-- Display name for the metric as used in the GUI -->
             <displayName>Interrupts</displayName>

             <!-- Brief description of the metric.. -->
             <description>Total number of system interrupts taken</description>

             <!-- The type of metric, used by the GUI to group metrics -->
             <type>interrupts</type>

             <!-- The colour to use for the metric graphs for this metric -->
             <colour>green</colour>
         </display>

     </metric>

     <!-- Metric group for interrupt metrics, used in the GUI -->
     <metricGroup id="Custom1">
         <!-- Display name for the group as use din the GUI -->
         <displayName>Custom1</displayName>

         <!-- Brief description of the group -->
         <description>Interrupt metrics</description>

         <!-- References to all the metrics included in the group -->
         <metric ref="com.allinea.metrics.custom1.interrupts"/>
     </metricGroup>

     <!-- Definition of the example source (metric plugin) used for the custom metric -->
     <source id="com.allinea.metrics.custom1_src">
         <!-- File name of the sample metric plugin shared library -->
         <sharedLibrary>libcustom1.so</sharedLibrary>
     </source>

</metricdefinitions>
```

9.5 report.xml

An example partial report definition file as detailed in Arm Performance Reports Integration. This informs Arm Performance Reports of the custom metrics implemented in custom1.c.

```
<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>
```

Index