



IST-2001-32133

GridLab - A Grid Application Toolkit and Testbed

## D11.3 GRID MONITORING ARCHITECTURE PROTOTYPE

---

Author(s):	Zoltán Balaton, Gábor Gombás
Document Filename:	GridLab-11-D11.3-05-Prototype
Work package:	WP11
Partner(s):	GridWare, MU, SZTAKI, VU
Lead Partner:	SZTAKI
Config ID:	GridLab-11-D11.2-05-v1.0
Document classification:	IST

---

**Abstract:** This document is the reference manual for the prototype of the monitoring system. The document describes required software, installation, configuration and command line options of the tools included in the prototype release. The documentation of the application programming interfaces (APIs) for both clients and sensor modules are also included.

## Contents

<b>1</b>	<b>Installation</b>	<b>2</b>
1.1	Software Requirements . . . . .	2
1.2	Building the Grid Monitoring System . . . . .	4
1.3	Configuration Files . . . . .	5
<b>2</b>	<b>Tools</b>	<b>7</b>
2.1	Example Consumers . . . . .	7
2.2	Producer Daemons . . . . .	10
<b>3</b>	<b>Monitoring System Protocol</b>	<b>12</b>
<b>4</b>	<b>Common Functions</b>	<b>14</b>
4.1	Buffer Handling . . . . .	14
4.2	Common functions . . . . .	21
<b>5</b>	<b>Consumer API</b>	<b>43</b>
<b>6</b>	<b>Producer API</b>	<b>54</b>
6.1	Producer loadable module support . . . . .	54
6.2	Producer library . . . . .	56
6.3	Common producer functions . . . . .	57
6.4	Producer data structures . . . . .	60

# 1 Installation

## 1.1 Software Requirements

This section lists the software components needed to build or develop the Grid Monitoring System. "Min. version" means earlier versions are known not to work. "Recommended version" means the version used during the development of the Grid Monitoring System; other versions might or might not work. The following software has to be installed to build the package:

- pkg-config
  - Min. version: 0.14.0
  - Recommended version: 0.14.0
  - Available at: <http://www.freedesktop.org/software/pkgconfig>
  - License: GPL
  
- GLib 2
  - Min. version: 1.3.15
  - Recommended version: 2.2.0
  - Available at: <ftp://ftp.gtk.org/pub/gtk>
  - License: LGPL-2

You may need the following software instead your platform's default if you experience problems during the build process:

- GCC
  - Recommended versions: 2.95, 3.2.1
  - Available at: <ftp://ftp.gnu.org/pub/gnu/gcc>
  - License: GPL-2
  
- GNU Make
  - Recommended version: 3.79.1
  - Available at: <ftp://ftp.gnu.org/pub/gnu/make>
  - License: GPL-2

The following software might need to be installed if you intend to modify the software:

- GNU autoconf
  - Min. version: 2.52g
  - Recommended version: 2.57
  - Available at: <ftp://ftp.gnu.org/pub/gnu/autoconf>
  - License: GPL-2
  
- GNU automake

- Min. version: 1.7
- Available at: <ftp://ftp.gnu.org/pub/gnu/automake>
- License: GPL-2
  
- GNU libtool
  - Recommended version: 1.4.3
  - Available at: <ftp://ftp.gnu.org/pub/gnu/automake>
  - License: GPL-2
  
- flex
  - Recommended version: 2.5.4a
  - Available at: <ftp://ftp.gnu.org/pub/non-gnu/flex>
  - License: BSD-like
  
- bison
  - Recommended versions: 1.35, 1.875
  - Available at: <ftp://ftp.gnu.org/pub/gnu/bison>
  - License: GPL-2

The following software is needed to build the documentation:

- gtk-doc-tools
  - Min. version: 0.6
  - Recommended version: 0.10
  - Available at: <ftp://ftp.gtk.org/pub/gtk-doc>
  - License: GPL
  
- Jade
  - Min. version: 1.1
  - Recommended version: 1.2.1
  - Available at: <http://www.jclark.com/jade>
  - License: Free
  
- libxslt
  - Recommended version: 1.0.21
  - Available at: <http://xmlsoft.org>
  - License: Free
  
- libxml2
  - Recommended version: 2.5.0

- Available at: <ftp://ftp.gnome.org>
- License: LGPL
- DocBook XML DTD
  - Required version: 4.1.2
  - Available at: <http://www.oasis-open.org/docbook>
  - License: Free
- DocBook Stylesheets
  - Recommended version: 1.76
  - Available at: <http://nwalsh.com/docbook/dsssl>
  - License: Free

Software needed to build the database archiver client:

- PostgreSQL
  - Min. version: 7.2
  - Recommended version: 7.3.1
  - Available at: <http://www.postgresql.org>
  - License: BSD

Software needed to build the simple loadavg monitor:

- Gtk+ 2
  - Min. version: 1.3.15
  - Recommended version: 2.0.0
  - Available at: <ftp://ftp.gtk.org/pub/gtk>
  - License: LGPL-2

## 1.2 Building the Grid Monitoring System

This section describes how to compile and install the Grid Monitoring System.

### Compiling

The Grid Monitoring System uses the GNU build utilities (autoconf, automake and libtool). The basic sequence for compilation and installation:

```
./configure
make
make install
```

## Configuration Options

The `./configure --help` command lists all available configuration options. Please see the autoconf documentation for information about the standard options.

The `configure` script for the Grid Monitoring System supports the following extra options:

```
configure [-with-gridlab] [-enable-gtk-doc] [-with-html-dir=DIR]
          [-enable-module-* | -disable-module-*] [-with-dmalloc]
          [-with-modedir=DIR] [-enable-java] [-with-java-includes=DIR]
```

### **--with-gridlab**

Use this option if building the package in a GridLab environment. This option sets the default `--prefix` to `$GRIDLAB_LOCATION/monitor-1.0.0` and enables runtime checking for the `GRIDLAB_LOCATION` environment variable in the consumer library.

### **--enable-gtk-doc**

Enables building the HTML documentation. Requires the `gtk-doc-tools` package to be installed.

### **--with-gridlab**

Where to install the HTML documentation.

### **--enable-module-\*, --disable-module-\***

Enable or disable building the specified module.

### **--with-dmalloc**

Use the `DMalloc` library for dynamic memory allocation debugging.

### **--with-modedir=DIR**

Where to install loadable modules.

### **--enable-java**

Enable Java support. Note: Java support is still under development and does not work currently.

### **--with-java-includes=DIR**

Where to find `<jni.h>`.

## 1.3 Configuration Files

In this section the format and common parts of configuration files are described.

### File format

Monitoring system configuration files consist of one or more sections. Sections consist of a name, an optional title string between brackets (`'[', '']`), and the section body between angle brackets (`'{', '}'`). A hashmark (`#`) indicates that the rest of the line is a comment. Example:

```
Common {
# This is the section named "Common"
}

Module["load.so"] {
# This is a section named "Module" with title "load.so"
}
```

Section names are case-independent, section titles are case-sensitive.

A section body might contain colon-terminated (':') variable definitions or other sections. A variable definition is a name-value pair separated by whitespace. Variable names must start with an ASCII letter and may contain letters and numbers. Variable names are case-independent. The supported value types:

- Integer numbers.
- Strings between double apostrophes (''). An apostrophe inside the string itself must be quoted by a backslash ('\'), a backslash must be quoted by another backslash.
- Boolean values: "yes" or "true" meaning true, "no" or "false" meaning false.

#### Example 1.1: Sample configuration fragment

```
Common {
ModulePath "/usr/lib/monitor";
AutoLoad no;
}

Module["hostsensor.so"] {
Priority 10;
}
```

There is also a compact format for specifying a variable inside a section:

```
Common::ModulePath "/usr/lib/monitor";
Module["hostsensor.so"]::Priority 10;
```

## Sections

### Common

The Common section contains configuration information for both the consumer and producer libraries. The following values are understood:

- **ModulePath (string)**

The directory name where the consumer or producer library will look for loadable modules. Default value: the module installation directory specified at compile time.

- **AutoLoad (boolean)**

If set to yes, modules found in the ModulePath directory will be loaded even if they are not mentioned in the configuration. If set to no, modules found are loaded only if there is a Module section with the same title as the name of the module, and it explicitly requests loading the module. Default value: true for the consumer library, false for the producer library.

## Module

The `Module` sections contain configuration information for loadable modules. The only commonly recognized variable is the `Load` boolean value which specifies whether the module should be loaded or not. If there is no `Load` variable, the value of `Common::AutoLoad` decides whether the module will be loaded or not.

Sensor modules have one more common variable called `Priority`. If more than one sensor modules provide the same metric, the one with the highest priority will be used. All other variables are interpreted by the module itself.

## Daemon

The daemon section contains configuration parameters for producer daemons. The following variables are understood:

- **PidFile (string)**

The name of the file where the daemon will store its process id.

- **LogTarget (string)**

Where to send log messages. The following formats are accepted:

- `SYSLOG:[facility]`: Send messages to syslog. If facility is missing, "daemon" is used.
- `STDERR::`: Send messages to the standard error.
- `FILE:filename`: Append messages to filename.

- **LogLevel (string)**

Only messages with a level of at least `LogLevel` will be logged. Possible values: `debug`, `notice`, `info`, `warning`, `err`.

- **LogIdent (string)**

Identity string to use in log messages.

- **Listen (string)**

URL to listen on for requests.

## 2 Tools

### 2.1 Example Consumers

This section presents the example consumer clients included in the prototype release.

#### testclient

A very simple consumer which can query one value of a given metric.

Usage:

```
testclient [-h] -n <metric> [-p <name=value>...] [-t] URL
```

**-h**

Print a short help message on parameters and usage.

**-n <name>**

Specify metric name to be queried. This option is mandatory.

**-p <name=value>**

Specify metric parameters. Each occurrence of this option adds a parameter to the list.

**-t**

Do timing for a simple latency measurement.

**URL**

URL of the monitoring system to connect to.

**archiver**

A simple archiver consumer which writes raw data to a file that can be read by **mcat**. The program can be stopped by pressing Ctrl-C (sending SIGINT).

Usage:

```
archiver [-h] -n <metric> [-p <name=value>...] -o <file> [-f  
    <freq>] URL
```

**-h**

Print a short help message on parameters and usage.

**-n <name>**

Specify metric name to be queried. This option is mandatory.

**-p <name=value>**

Specify metric parameters. Each occurrence of this option adds a parameter to the list.

**-o <file>**

Output file name to save data into. This option is mandatory.

**-f <freq>**

Specify measurement frequency in seconds.

**URL**

URL of the monitoring system to connect to.

**mcat**

Display contents of a file created by **archiver**.

Usage:

```
mcat [-f] [-i <metric_id>] [-e <expression>] URL
```

**-f**

Ignore EOF and continue reading the file.

**-i <metric\_id>**

Display data only with the specified metric ID.

**-e <expression>**

Evaluate the expression on every data and print the result. If given, the **-i** option should be specified as well.

**file**

Input file name to read data from.

## monclient

A generic consumer client combining the features of **testclient**, **archiver** and **mcat**.

Usage:

```
monclient [-h] [-c <count>] [-e <expr>] [-f <freq>] -n <metric>
          [-p <name=value>...] [-o <file>] [-r] [-v] URL
```

### -h

Print a short help message on parameters and usage.

### -c <count>

Receive number of values specified (default is 1) and then exit. If set to negative the program runs until stopped by pressing Ctrl-C (sending SIGINT).

**-e <expression>** Evaluate the expression on every data and print the result.

### -f <freq>

Specify measurement frequency in seconds.

### -n <name>

Specify metric name to be queried. This option is mandatory.

### -p <name=value>

Specify metric parameters. Each occurrence of this option adds a parameter to the list.

### -o <file>

Output file name to save data into.

### -r

Output raw data. If given, the `-o` option must also be specified. This option cannot be used together with the `-e` option. The saved output can be viewed with **mcat**.

### URL

URL of the monitoring system to connect to.

### -v

Verbose output, some additional control information is printed as well.

## dbarchiver

An archiver consumer that stores data received in a PostgreSQL database.

Usage:

```
dbarchiver [-h] -c <file>
```

### -h

Print a short help message on parameters and usage.

### -c <file>

Use the specified configuration file.

The configuration file has the syntax described [here](#). The following main sections are recognized:

- **Database**

Defines the database connection parameters. The following values can be set:

- **Host (string)** The host name where the database is located.
- **Name (string)** The name of the database to connect to.
- **User (string)** The user to connect as.
- **Password (string)** The password to use for connecting to the database.

- **Producer**

A `Producer` section defines a producers to connect to. The section's title should be the producer's URL. A `Producer` section may contain one or more `Metric` subsections defining the metrics to monitor. The title of a `Metric` section should be the name of the metric. `Metric` sections may contain the following:

- **Param subsections**

The title of a `Param` subsection should be the name of a metric parameter. The only allowed variable inside a `Param` subsection is `Value`, which defines the value of the metric parameter. Currently `Value` must be of string type regardless the actual type of the metric parameter; it will likely change future releases.

- Query (string)**

The query variable defines the database query (normally an `INSERT SQL` statement) to perform when a metric value of the defined metric arrives. It is a string with possible substitution macros. These macros have the format of `%{expr}` where `expr` is a valid expression for the data type of the defined metric, or one of the following special values:

- \* `timestamp`: The metric value's timestamp in ISO format
- \* `unixtime`: The metric value's timestamp in UNIX time format

## **vis**

A very simple visualisation example consumer. It reads `host.loadavg` metrics in a format as outputted by `monclient` from its standard input and draws a graph of it in a window.

## **2.2 Producer Daemons**

This section describes the daemon processes implementing the monitoring system producers.

### **mm**

The Main Monitor daemon.

Usage:

```
mm [-c <configfile>] [-d] [-h <hostname>]
```

**-c <configfile>**

Use the specified file as the configuration file instead of the default (`SYSCONFDIR/mm.conf`).

**-d**

Do not fork to the background and log to the standard error output instead of the logfile specified in the configuration.

**-h <hostname>**

Use the specified canonical hostname instead of the default.

## Example 2.1: Sample dbarchiver configuration

```
Database {
Host "localhost";
Name "mydb";
User "myuser";
Password "mypasswd";
}

Producer["monp://localhost"] {
Metric["host.loadavg"] {
Param["host"]::Value "my.fqdn";
Param["frequency"]::Value "5";
Query "INSERT INTO loadavg (ts, l1, l5, l15) VALUES ({unixtime}, {val[0]}, {v
}

Metric["host.mem.avail"] {
Param["host"]::Value "my.fqdn";
Param["frequency"]::Value "5";
Query "INSERT INTO memavail (ts, value) VALUES ({unixtime}, {val})";
}

Metric["host.cpu.user"] {
Param["host"]::Value "my.fqdn";
Param["frequency"]::Value "5";
Param["cpu"]::Value "0";
Query "INSERT INTO usercpu (ts, value) VALUES ({unixtime}, {val})";
}
}
```

**lm**

The Local Monitor daemon. Currently the only hard-coded difference from the **mm** is that the **lm** only accepts queries for host-specific metrics if the host parameter equals to the local hostname. All other differences are specified in the configuration file.

Usage:

```
lm [-c <configfile>] [-d] [-h <hostname>]
```

**-c <configfile>**

Use the specified file as the configuration file instead of the default (SYSCONFDIR/lm.conf).

**-d**

Do not fork to the background and log to the standard error output instead of the logfile specified in the configuration.

**-h <hostname>**

Use the specified canonical hostname instead of the default.

### 3 Monitoring System Protocol

This section is an overview of the Consumer-Producer protocol used in the Grid Monitoring System.

#### General

The three main protocol data units are commands, command responses and metric values. Every command has an identifier (e.g. COLLECT or GET) and a sequence number. Responses have an error code, the same sequence number as the original command and an optional result identifier. It is possible to issue multiple commands without waiting for a response for the previous command to arrive. In case of multiple parallel commands the protocol does not guarantee that the order of responses matches the order of the original commands.

The producer is allowed to send metric values at any time. The only restriction is that if a metric value is being sent as a result of a command, the response for the command must precede the metric value (i.e. when using the GET command to request data the response for the command must be sent before the requested data corresponding to this GET command).

A channel is a logical connection between a producer and a consumer defined over a physical network connection. Each channel has a channel identifier which is assigned by, and specific to, the producer. There are two special channel identifiers a consumer can use. One references all channels defined between it and the producer it is talking to, the other references the current channel. When a network connection is opened between a producer and a consumer, the producer automatically defines a channel and makes it the current channel. The consumer can later define new channels and change the current channel.

#### Commands

The consumer can use the following commands:

##### **AUTH**

- Arguments: authentication method, method-specific parameters
- Response: error code, channel identifier

The AUTH command authenticates the user to the monitoring system. The AUTH command should be used once and it should be the first command issued by the consumer after a channel is opened. Any other commands issued before the AUTH command will result in an error. Issuing the AUTH command on an already authenticated channel will also result in an error. If the authentication was successful an identifier for the current channel will be returned.

##### **COLLECT**

- Arguments: metric name, parameter list
- Response: error code, metric identifier

The COLLECT command instructs the monitoring system to create a metric instance with the given parameters. If this is successful, the response contains the metric identifier of the metric instance. The parameter list is a list of name/value pairs.

##### **STOP**

- Arguments: metric identifier, channel identifier
- Response: error code

The STOP command tells the monitoring system that no more metric values for this identifier should be sent to the specified channel. Metric values for this identifier that have been queued in the monitoring system for the specified channel will be lost.

If the metric identifier is still subscribed to other channels, both the GET and SUBSCRIBE commands can be used to receive further metric values for this metric id. If there are no other channels where this metric identifier is subscribed, the STOP command will destroy the metric instance and all further references to this metric instance will result in an error.

#### **SUBSCRIBE**

- Arguments: metric identifier, channel identifier
- Response: error code

The SUBSCRIBE command instructs the monitoring system that all metric values for the given metric identifier should be automatically sent to the specified channel. The same metric identifier can be subscribed to more than one channel by using the SUBSCRIBE command multiple times. If there were metric values queued for the given metric identifier on the current channel they will be copied to the destination channel.

#### **BUFFER**

- Arguments: metric identifier, channel identifier
- Response: error code

The BUFFER command instructs the monitoring system that all metric values for the given metric identifier should be buffered on the specified channel. It is possible to use the SUBSCRIBE and BUFFER commands for the same metric identifier on different channels.

#### **GET**

- Arguments: metric identifier, channel identifier
- Response: error code

The GET command performs the following tasks:

- If there are metric values queued for the given metric identifier on the given channel, the monitoring system will send them to the consumer.
- For continuous metrics, a new measurement is requested from the appropriate sensor. When the measurement is ready, the measured metric value will be sent to the consumer on the given channel. The monitoring system makes no guarantee when (if ever) this metric value will be sent.

#### **GET\_SPECIAL**

- Arguments: metric identifier, parameter
- Response: error code

The GET\_SPECIAL command is similar to the **GET** command and is used to request specific internal data from the monitoring system (e.g. metric definitions). The metric identifier should be a special predefined metric identifier, it cannot be an identifier returned by the COLLECT command. The parameter is either a metric or a channel identifier depending on the first argument. This command is meant for internal communication between the consumer API and the producer and may change in later protocol versions.

## QUERY

- Arguments: metric name, parameter list
- Response: error code, metric identifier

The QUERY command is a fast way for retrieving one metric value. It is functionally equivalent to a COLLECT, GET, STOP sequence.

## 4 Common Functions

This sections contains documentation for functions and utilities common to the consumer and the producer.

### 4.1 Buffer Handling

#### Description

Buffer handling functions

#### Details

##### **BUFFER\_BLOCK\_SIZE**

```
#define BUFFER_BLOCK_SIZE 64
```

This macro defines the size of chunks in which memory will be allocated. Must be a power of 2.

##### **buf\_init ()**

```
void          buf_init                (mon_buffer *buf);
```

Initialize a buffer.

*buf* : a **mon\_buffer** to initialize.

##### **buf\_destroy ()**

```
void          buf_destroy              (mon_buffer *buf);
```

Destroy the contents of a buffer.

*buf* : a **mon\_buffer** to destroy.

### **buf\_ensure\_free ()**

```
int          buf_ensure_free          (mon_buffer *buf,  
                                     size_t len);
```

Check that the buffer *buf* is big enough to hold at least *len* bytes of additional data. If the buffer is smaller then it is enlarged to ensure this.

*buf* : a **mon\_buffer** to be checked/enlarged.

*len* : number of free bytes requested.

**Returns** : 0 or -1 if memory allocation failed.

### **buf\_normalize ()**

```
void          buf_normalize           (mon_buffer *buf);
```

Normalize the contents of a buffer, move used data to the beginning.

*buf* : a **mon\_buffer** to normalize.

### **buf\_copy ()**

```
int          buf_copy                 (mon_buffer *dst,  
                                     mon_buffer *src);
```

Copy the contents of the buffer *src* to buffer *dst*. Buffer *dst* is enlarged appropriately if it is too small.

*dst* : a **mon\_buffer** to hold copied data.

*src* : a **mon\_buffer** to copy.

**Returns** : 0 or error code. If memory allocation failed ENOMEM is returned.

### **buf\_copy\_data ()**

```
int          buf_copy_data           (mon_buffer *buf,  
                                     void *dst,  
                                     size_t len);
```

Copy *len* bytes of raw data from the start of buffer *buf* to *dst*.

*buf* : a **mon\_buffer** to copy data from.

*dst* : pointer to the start of the memory to hold copied data. Must be big enough to hold at least *len* bytes.

*len* : number of bytes to copy.

**Returns** : 0 or error code. If the buffer has less than *len* bytes of data EINVAL is returned.

### **buf\_add\_int32 ()**

```
void          buf_add_int32                (mon_buffer *buf,  
                                           int32_t val);
```

Append the signed 32 bit integer value *val* to buffer *buf*. The buffer must have enough free space to hold the value.

*buf* : a **mon\_buffer** to add data to.

*val* : the value to be added.

### **buf\_add\_uint32 ()**

```
void          buf_add_uint32              (mon_buffer *buf,  
                                           uint32_t val);
```

Append the unsigned 32 bit integer value *val* to buffer *buf*. The buffer must have enough free space to hold the value.

*buf* : a **mon\_buffer** to add data to.

*val* : the value to be added.

### **buf\_add\_int64 ()**

```
void          buf_add_int64                (mon_buffer *buf,  
                                           int64_t val);
```

Append the signed 64 bit integer value *val* to buffer *buf*. The buffer must have enough free space to hold the value.

*buf* : a **mon\_buffer** to add data to.

*val* : the value to be added.

### **buf\_add\_uint64 ()**

```
void          buf_add_uint64              (mon_buffer *buf,  
                                           uint64_t val);
```

Append the unsigned 64 bit integer value *val* to buffer *buf*. The buffer must have enough free space to hold the value.

*buf* : a **mon\_buffer** to add data to.

*val* : the value to be added.

### **buf\_add\_double ()**

```
void          buf_add_double          (mon_buffer *buf,
                                       double val);
```

Append the double precision (64 bit) floating point value *val* to buffer *buf*. The buffer must have enough free space to hold the value.

*buf* : a **mon\_buffer** to add data to.

*val* : the value to be added.

### **buf\_add\_string ()**

```
void          buf_add_string          (mon_buffer *buf,
                                       const char *val);
```

Append the string value *val* to buffer *buf*. The buffer must have enough free space to hold the value.

*buf* : a **mon\_buffer** to add data to.

*val* : pointer to a \0 terminated C string to be added.

### **buf\_add\_opaque ()**

```
void          buf_add_opaque          (mon_buffer *buf,
                                       const void *val,
                                       uint32_t len);
```

Append *len* number of bytes opaque data starting at *val* to buffer *buf*. The buffer must have enough free space to hold the value.

*buf* : a **mon\_buffer** to add data to.

*val* : pointer to the data to be added.

*len* : number of bytes to be added.

### **buf\_get\_int32 ()**

```
int          buf_get_int32           (mon_buffer *buf,
                                       int32_t *val);
```

Get data from the beginning of the buffer *buf* converted as a signed 32 bit integer. The value is put in *val* and removed from the buffer.

*buf* : a **mon\_buffer** to get data from.

*val* : the value returned.

**Returns** : 0 or -1 if the buffer has not enough data.

### **buf\_get\_uint32 ()**

```
int          buf_get_uint32                (mon_buffer *buf,  
                                           uint32_t *val);
```

Get data from the beginning of the buffer *buf* converted as an unsigned 32 bit integer. The value is put in *val* and removed from the buffer.

*buf* : a **mon\_buffer** to get data from.

*val* : the value returned.

**Returns** : 0 or -1 if the buffer has not enough data.

### **buf\_get\_int64 ()**

```
int          buf_get_int64                (mon_buffer *buf,  
                                           int64_t *val);
```

Get data from the beginning of the buffer *buf* converted as a signed 64 bit integer. The value is put in *val* and removed from the buffer.

*buf* : a **mon\_buffer** to get data from.

*val* : the value returned.

**Returns** : 0 or -1 if the buffer has not enough data.

### **buf\_get\_uint64 ()**

```
int          buf_get_uint64              (mon_buffer *buf,  
                                           uint64_t *val);
```

Get data from the beginning of the buffer *buf* converted as an unsigned 64 bit integer. The value is put in *val* and removed from the buffer.

*buf* : a **mon\_buffer** to get data from.

*val* : the value returned.

**Returns** : 0 or -1 if the buffer has not enough data.

### **buf\_get\_double ()**

```
int          buf_get_double              (mon_buffer *buf,  
                                           double *val);
```

Get data from the beginning of the buffer *buf* converted as a double precision (64 bit) floating point value. The value is put in *val* and removed from the buffer.

*buf* : a **mon\_buffer** to get data from.

*val* : the value returned.

**Returns** : 0 or -1 if the buffer has not enough data.

### **buf\_get\_string ()**

```
int          buf_get_string          (mon_buffer *buf,
                                     char **s,
                                     size_t *slen);
```

Get data from the beginning of the buffer *buf* converted as a string. The string is copied to memory allocated using `malloc` and removed from the buffer. Pointer to the string is returned in *s* and the length of the string is returned in *slen* if it is not NULL.

*buf* : a `mon_buffer` to get data from.

*s* : returned pointer to a `\0` terminated C string allocated using `malloc`

*slen* : length of string returned if not NULL

**Returns** : 0 or -1 if the buffer has not enough data or memory allocation failed.

### **buf\_get\_opaque ()**

```
int          buf_get_opaque         (mon_buffer *buf,
                                     void **s,
                                     size_t *slen);
```

Get opaque data bytes from the beginning of the buffer *buf*. The data is copied to memory allocated using `malloc` and removed from the buffer. Pointer to the data is returned in *val* and the length of the string is returned in *slen*.

*buf* : a `mon_buffer` to get data from.

*s* : returned pointer to memory allocated using `malloc` holding the data

*slen* : length of data returned. Must not be NULL.

**Returns** : 0 or -1 if the buffer has not enough data or memory allocation failed.

### **buf\_peek\_int32 ()**

```
int          buf_peek_int32         (mon_buffer *buf,
                                     int32_t *val);
```

Peek data at the beginning of the buffer *buf* converted as a signed 32 bit integer. The value is returned in *val* and also kept in the buffer.

*buf* : a `mon_buffer` to peek data in.

*val* : the value returned.

**Returns** : 0 or -1 if the buffer has not enough data.

### **buf\_peek\_uint32 ()**

```
int          buf_peek_uint32          (mon_buffer *buf,  
                                     uint32_t *val);
```

Peek data at the beginning of the buffer *buf* converted as an unsigned 32 bit integer. The value is returned in *val* and also kept in the buffer.

*buf* : a **mon\_buffer** to peek data in.

*val* : the value returned.

**Returns** : 0 or -1 if the buffer has not enough data.

### **buf\_read ()**

```
int          buf_read                 (mon_buffer *buf,  
                                     int sd);
```

Read data from the specified file descriptor *sd* into a buffer *buf*.

*buf* : a **mon\_buffer** to read data into.

*sd* : a file descriptor to read data from.

**Returns** : 0 or error code.

### **buf\_write ()**

```
int          buf_write                (mon_buffer *buf,  
                                     int sd);
```

Write the contents of the buffer *buf* to the specified file descriptor *sd*.

*buf* : a **mon\_buffer** to write contents of.

*sd* : a file descriptor to write data to.

**Returns** : 0 or error code.

### **buf\_write\_all ()**

```
int          buf_write_all            (mon_buffer *buf,  
                                     int sd);
```

Write the contents of the buffer *buf* to the specified file descriptor *sd*. Like **buf\_write()**, but does not return on partial write.

*buf* : a **mon\_buffer** to write contents of.

*sd* : a file descriptor to write data to.

**Returns** : 0 or error code.

## 4.2 Common functions

### Description

Functions and utilities common to the consumer and the producer

### Details

#### **MON\_METRIC\_ID\_MIN**

```
#define MON_METRIC_ID_MIN 256
```

This macro defines the smallest value the producer is allowed to allocate to normal metric IDs.

#### **MON\_METRIC\_ID\_MAX**

```
#define MON_METRIC_ID_MAX ((1 << 24) - 1)
```

This macro defines the largest value the producer is allowed to allocate to normal metric IDs.

#### **MID\_CMD\_RESPONSE**

```
#define MID_CMD_RESPONSE 0
```

Predefined metric identifier for command responses.

#### **MID\_PRODUCER\_CAPS**

```
#define MID_PRODUCER_CAPS 1
```

Predefined metric identifier for producer capabilities.

#### **MON\_IO\_READ**

```
#define MON_IO_READ (1 << 0)
```

Macro meaning a file descriptor is ready for reading.

#### **MON\_IO\_WRITE**

```
#define MON_IO_WRITE (1 << 1)
```

Macro meaning a file descriptor is ready for writing.

### **MON\_IO\_ERROR**

```
#define MON_IO_ERROR (1 << 2)
```

Macro meaning a file descriptor has an error condition.

### **MON\_T\_INT32**

```
#define MON_T_INT32 1
```

The internal code for 32-bit signed integers.

### **MON\_T\_UINT32**

```
#define MON_T_UINT32 2
```

The internal code for 32-bit unsigned integers.

### **MON\_T\_INT64**

```
#define MON_T_INT64 3
```

The internal code for 64-bit signed integers.

### **MON\_T\_UINT64**

```
#define MON_T_UINT64 4
```

The internal code for 64-bit unsigned integers.

### **MON\_T\_DOUBLE**

```
#define MON_T_DOUBLE 5
```

The internal code for 64-bit double precision numbers.

### **MON\_T\_STRING**

```
#define MON_T_STRING 6
```

The internal code for the string data type.

### **MON\_T\_OPAQUE**

```
#define MON_T_OPAQUE 7
```

The internal code for opaque data.

### **MON\_T\_REC**

```
#define MON_T_REC 8
```

The internal code for record data types.

### **MON\_T\_ARRAY**

```
#define MON_T_ARRAY 9
```

The internal code for array data types.

### **MON\_T\_BOOLEAN**

```
#define MON_T_BOOLEAN 10
```

The internal code for booleans.

### **MON\_T\_MAX\_TYPE**

```
#define MON_T_MAX_TYPE MON_T_BOOLEAN
```

The largest data type value.

### **MON\_CFG\_SECTION**

```
#define MON_CFG_SECTION (MON_T_MAX_TYPE + 1)
```

Data type for configuration sections.

### **MON\_CFG\_LEAF**

```
#define MON_CFG_LEAF (MON_T_MAX_TYPE + 2)
```

Data type for configuration leaf values.

### **MON\_TYPE\_MULTIPLE**

```
#define MON_TYPE_MULTIPLE (1 << 0)
```

A string representation is allowed to contain multiple types.

### **MON\_TYPE\_SIMPLE**

```
#define MON_TYPE_SIMPLE (1 << 1)
```

A string representation is only allowed to contain non-aggregate types.

### **MON\_PARAM\_OPTIONAL**

```
#define MON_PARAM_OPTIONAL (1 << 0)
```

Specifies that the parameter is optional.

### **MON\_PARAM\_MULTIPLE**

```
#define MON_PARAM_MULTIPLE (1 << 1)
```

Specifies that the parameter might have multiple values.

### **mon\_log\_callback ()**

```
void (*mon_log_callback) (int level,  
                          const char *message);
```

Specifies the type of the function which is passed to `mon_log_set()`. This function is passed a log level and a message.

*level* : the log level.

*message* : the log message.

### **mon\_module\_scan\_callback ()**

```
int (*mon_module_scan_callback) (void *handle,  
                                  const char *name,  
                                  void **entry_points,  
                                  void *cb_arg);
```

Specifies the type of the function which is passed to `mon_module_scan()`. This function is passed the module object's handle, the file name (without path information), the values of the symbols specified to `mon_module_scan()` and an opaque pointer passed to `mon_module_scan()`.

*handle* : the shared object's handle.

*name* : the file name.

*entry\_points* : the values of the requested symbols.

*cb\_arg* : an opaque pointer passed to `mon_module_scan()`.

**Returns** : 0 if the module is successfully initialized. If the return value is non-0, the module will be unloaded.

**mon\_io\_callback ()**

```
int (*mon_io_callback) (int fd,
                        unsigned event,
                        void *ptr);
```

Specifies the type of the function which is passed to `mon_io_set_callback()`.

This function is passed the file descriptor which caused an event, an event mask and an opaque pointer originally passed to `mon_io_set_callback()`.

*fd* : the file descriptor having an outstanding event.

*Param2* : the event mask.

*ptr* : the opaque pointer passed to `mon_io_set_callback()`.

**Returns** : 0 if the event was processed, -1 if the file descriptor was closed.

**MON\_DOM\_METRIC**

```
#define MON_DOM_METRIC 1
```

Specifies that a name is a metric.

**MON\_DOM\_CTRL**

```
#define MON_DOM_CTRL 2
```

Specifies that a name is a control.

**mon\_metric\_lookup\_func ()**

```
mon_metric_def* (*mon_metric_lookup_func) (const char *name,
                                           int domain);
```

Specifies the type of the function which is passed to `mon_registry_register()`.

This function is passed a name and the domain (which is either `MON_DOM_METRIC` or `MON_DOM_CTRL`).

*name* : the name of the metric or control.

*domain* : the domain.

**Returns** : the metric (control) definition or NULL if the name was invalid.

**MON\_RESP\_OK**

```
#define MON_RESP_OK 0
```

Monitor protocol response code for no error.

### **MON\_RESP\_UNKNOWN\_CMD**

```
#define MON_RESP_UNKNOWN_CMD 1
```

Monitor protocol response code for unknown commands.

### **MON\_RESP\_UNKNOWN\_METRIC**

```
#define MON_RESP_UNKNOWN_METRIC 2
```

Monitor protocol response code for unknown metrics.

### **MON\_RESP\_UNKNOWN\_CHAN**

```
#define MON_RESP_UNKNOWN_CHAN 3
```

Monitor protocol response code for unknown channels.

### **MON\_RESP\_BADPARAM**

```
#define MON_RESP_BADPARAM 4
```

Monitor protocol response code for bad parameters.

### **MON\_RESP\_AUTH\_NEEDED**

```
#define MON_RESP_AUTH_NEEDED 5
```

Monitor protocol response code when authentication is needed before continuing.

### **MON\_RESP\_AUTH\_ERR**

```
#define MON_RESP_AUTH_ERR 6
```

Monitor protocol response code for authentication error.

### **MON\_RESP\_GENERIC\_ERR**

```
#define MON_RESP_GENERIC_ERR 7
```

Monitor protocol response code for generic error.

### **MON\_RESP\_RES\_LIMIT**

```
#define MON_RESP_RES_LIMIT 8
```

Monitor protocol response code for exceeding resource limits.

**mon\_type\_parse ()**

```
int          mon_type_parse          (const char *typedesc,
                                     unsigned parse_flags,
                                     mon_type **type);
```

Create a type descriptor from the given string representation.

*typedesc* : textual type description

*Param2* : parsing options. Can be the logical OR of MON\_TYPE\_MULTIPLE and MON\_TYPE\_SIMPLE

*type* : contains the parsed type on return

**Returns** : 0 or error code

**mon\_type\_parse\_simple ()**

```
int          mon_type_parse_simple   (const char *type);
```

Parse a simple type.

*type* : string representation of a single type.

**Returns** : code of the simple type or -1 on error.

**mon\_type\_done ()**

```
void        mon_type_done           (mon_type *type);
```

Drops reference on a **mon\_type**. When the last reference is gone the memory allocated to the type is freed.

*type* : a **mon\_type**.

**mon\_type\_tostr ()**

```
int          mon_type_tostr          (char *dst,
                                     int maxlen,
                                     mon_type *type);
```

Converts a **mon\_type** to a string representation.

*dst* : output buffer.

*maxlen* : size of the output buffer.

*type* : a **mon\_type** to stringify.

**Returns** : 0 or error code. If the buffer was too small, ERANGE is returned.

### **mon\_type\_tostr\_simple ()**

```
const char* mon_type_tostr_simple (int type);
```

Converts a simple type to a string representation.

*type* : a type code (one of the MON\_T\_ constants).

**Returns** : the string representation of the type or NULL on error.

### **mon\_expr\_parse ()**

```
int mon_expr_parse (const char *expr,
                   mon_type *type,
                   mon_buffer *buf,
                   mon_type **etype);
```

Parse the expression *expr* on the buffer *buf* containing data of type *type*. The type of the expression (which is a subtype of *type*) will be returned in *etype*. The buffer's **start** pointer is updated to point to the beginning of the expression's data.

*expr* : expression to parse.

*type* : data type of the input buffer.

*buf* : a **mon\_buffer** containing the input data.

*etype* : on return contains the type of the expression.

**Returns** : 0 or error code.

### **mon\_expr\_compile ()**

```
int mon_expr_compile (const char *expr,
                    mon_type *type,
                    mon_compiled_expr **prog,
                    mon_type **etype);
```

Compile an expression to an internal form that can be used if the expression is to be evaluated several times.

*expr* : expression to compile.

*type* : data type of the input buffer.

*prog* : on return contains the compiled expression.

*etype* : on return contains the type of the expression.

**Returns** : 0 or error code.

**mon\_expr\_print ()**

```
int          mon_expr_print          (char *dst,  
                                     int maxlen,  
                                     mon_buffer *buf,  
                                     mon_type *type);
```

Create a textual representation of *buf* in the string *dst*.

*dst* : the output buffer.

*maxlen* : length of the output buffer.

*buf* : a **mon\_buffer** containing the input data.

*type* : the type of *buf*->start.

**Returns** : 0 or error code. If the buffer was too small, ERANGE is returned.

**mon\_compiled\_expr\_execute ()**

```
int          mon_compiled_expr_execute (mon_compiled_expr *prog,  
                                       mon_buffer *buf);
```

Execute a previously compiled expression. It updates *buf*->start to point to the beginning of the expression.

*prog* : the compiled expression.

*buf* : the input buffer.

**Returns** : 0 or error code.

**mon\_compiled\_expr\_free ()**

```
void          mon_compiled_expr_free (mon_compiled_expr *prog);
```

Free the memory allocated to a compiled expression.

*prog* : a compiled expression.

**mon\_metric\_def\_get ()**

```
mon_metric_def* mon_metric_def_get (const char *name);
```

Get the definition of the named metric. The result of the first successful registered lookup function will be returned.

*name* : the name of the metric.

**Returns** : a **mon\_metric\_def** or NULL if the metric name was invalid.

**mon\_ctrl\_def\_get ()**

```
mon_metric_def* mon_ctrl_def_get (const char *name);
```

Get the definition of the named control. The result of the first successful registered lookup function will be returned.

*name* : the name of the control/

**Returns** : a `mon_ctrl_def` or NULL if the control name was invalid.

**mon\_metric\_def\_done ()**

```
void mon_metric_def_done (mon_metric_def *md);
```

Drops a reference to a `mon_metric_def`. When the last reference is gone the memory allocated to the `mon_metric_def` is freed.

*md* : a `mon_metric_def`.

**mon\_registry\_register ()**

```
int mon_registry_register (mon_metric_lookup_func func);
```

Register a metric definition lookup function. Such functions will be called in the order of registration when a lookup for a metric definition is requested.

*func* : a lookup function to register.

**Returns** : 0 or error code.

**mon\_registry\_unregister ()**

```
void mon_registry_unregister (mon_metric_lookup_func func);
```

Unregister a metric definition lookup function.

*func* : the function to unregister.

**mon\_metric\_value\_new ()**

```
mon_metric_value* mon_metric_value_new (struct timeval *tv,  
void *id);
```

Allocates a new metric value.

*tv* : timestamp of the metric value.

*id* : identifier of the metric instance.

**Returns** : a new `mon_metric_value` or NULL on error.

**mon\_metric\_value\_free ()**

```
void      mon_metric_value_free      (mon_metric_value *mv);
```

Free the memory allocated to a metric value.

*mv* : a **mon\_metric\_value**.

**mon\_module\_close ()**

```
void      mon_module_close      (void *handle);
```

Unloads a loaded module.

*handle* : the handle of the module.

**mon\_module\_scan ()**

```
int      mon_module_scan      (const char *mod_path,
                               int num_sym,
                               const char **sym_names,
                               mon_module_scan_callback cb,
                               void *cb_arg);
```

Scan a directory for modules to load.

This function examines every shared object in the *mod\_path* directory and looks for the symbols named *sym\_names*. If all the requested symbols were found in the module, their values along are passed to the *cb* function along with the file name and the supplied *cb\_arg* pointer. If the callback function returns 0, the module is registered as loaded; otherwise, the module is unloaded.

*mod\_path* : the directory to scan.

*num\_sym* : number of symbols to look for.

*sym\_names* : symbol names to look for.

*cb* : a function of type **mon\_module\_scan\_callback** that will be called for every modules found.

*cb\_arg* : an opaque pointer to pass to the callback.

**Returns :****mon\_log\_set ()**

```
void      mon_log_set      (mon_log_callback cb);
```

Set the log callback function.

*cb* : the log callback function.

**mon\_log ()**

```
void          mon_log                (int level,  
                                     const char *fmt,  
                                     ...);
```

Log a message, similar to syslog.

*level* : log level.

*fmt* : format string.

*...* : format parameters.

**mon\_resp\_to\_err ()**

```
int          mon_resp_to_err        (uint32_t resp);
```

Transforms a protocol response code to an internal error code.

*resp* : protocol response code.

**Returns** : internal error code.

**mon\_err\_to\_resp ()**

```
uint32_t    mon_err_to_resp        (int err);
```

Transforms an internal error code to a protocol response code.

*err* : internal error code.

**Returns** : protocol response code.

**mon\_arg\_cmp ()**

```
int          mon_arg_cmp            (mon_arg *a1,  
                                     mon_arg *a2);
```

Compares two arguments.

*a1* : a **mon\_arg**.

*a2* : a **mon\_arg**.

**Returns** : less than zero, zero or greater than zero if *a1* is less than, equal to or greater than *a2*, respectively.

**mon\_arg\_free ()**

```
void          mon_arg_free          (mon_arg *a);
```

Frees the memory allocated to a **mon\_arg**.

*a* : a **mon\_arg**.

**mon\_arg\_list\_new ()**

```
mon_arg_list* mon_arg_list_new          (void);
```

Allocates a new **mon\_arg\_list**.

**Returns** : the allocated **mon\_arg\_list**.

**mon\_arg\_list\_free ()**

```
void          mon_arg_list_free          (mon_arg_list *l);
```

Frees every element of a **mon\_arg\_list** and frees the memory allocated to the list itself.

*l* : a **mon\_arg\_list**.

**mon\_arg\_list\_add ()**

```
int          mon_arg_list_add            (mon_arg_list *l,  
                                         mon_type *type,  
                                         mon_simple_types *val);
```

Add a new argument to a **mon\_arg\_list**.

*l* : a **mon\_arg\_list**.

*type* : a **mon\_type** describing the name and data type of the argument.

*val* : the value of an argument.

**Returns** : 0 if successful, ENOMEM when out of memory.

**mon\_arg\_list\_add\_string ()**

```
int          mon_arg_list_add_string     (mon_arg_list *l,  
                                         const char *name,  
                                         const char *value);
```

Add a string argument to a **mon\_arg\_list**.

*l* : a **mon\_arg\_list**.

*name* : the name of the argument.

*value* : the argument's value.

**Returns** : 0 if successful, ENOMEM when out of memory.

### **mon\_arg\_list\_add\_int32 ()**

```
int          mon_arg_list_add_int32          (mon_arg_list *l,  
                                             const char *name,  
                                             int32_t value);
```

Add a 32-bit signed integer to a **mon\_arg\_list**.

*l* : a **mon\_arg\_list**.

*name* : the name of the argument.

*value* : the argument's value.

**Returns** : 0 if successful, ENOMEM when out of memory.

### **mon\_arg\_list\_add\_uint32 ()**

```
int          mon_arg_list_add_uint32        (mon_arg_list *l,  
                                             const char *name,  
                                             uint32_t value);
```

Add a 32-bit unsigned integer to a **mon\_arg\_list**.

*l* : a **mon\_arg\_list**.

*name* : the name of the argument.

*value* : the argument's value.

**Returns** : 0 if successful, ENOMEM when out of memory.

### **mon\_arg\_list\_add\_int64 ()**

```
int          mon_arg_list_add_int64        (mon_arg_list *l,  
                                             const char *name,  
                                             int64_t value);
```

Add a 64-bit signed integer to a **mon\_arg\_list**.

*l* : a **mon\_arg\_list**.

*name* : the name of the argument.

*value* : the argument's value.

**Returns** : 0 if successful, ENOMEM when out of memory.

**mon\_arg\_list\_add\_uint64 ()**

```
int          mon_arg_list_add_uint64          (mon_arg_list *l,  
                                              const char *name,  
                                              uint64_t value);
```

Add a 64-bit unsigned integer to a **mon\_arg\_list**.

*l* : a **mon\_arg\_list**.

*name* : the name of the argument.

*value* : the argument's value.

**Returns** : 0 if successful, ENOMEM when out of memory.

**mon\_arg\_list\_add\_double ()**

```
int          mon_arg_list_add_double         (mon_arg_list *l,  
                                              const char *name,  
                                              double value);
```

Add a 64-bit double precision floating point number to a **mon\_arg\_list**.

*l* : a **mon\_arg\_list**.

*name* : the name of the argument.

*value* : the argument's value.

**Returns** : 0 if successful, ENOMEM when out of memory.

**mon\_arg\_list\_add\_parsed ()**

```
int          mon_arg_list_add_parsed         (mon_arg_list *l,  
                                              const char *name,  
                                              int type,  
                                              const char *value);
```

Parse an argument given as a string and append it to the *l* **mon\_arg\_list**.

This function is useful when the user specifies an argument in a textual format (e.g. command line).

*l* : a **mon\_arg\_list**.

*name* : the name of the argument.

*type* : the type of the argument.

*value* : the argument's value.

**Returns** : 0 if successful, ENOMEM when out of memory, EINVAL if the supplied *value* could not be parsed.

**mon\_arg\_list\_copy ()**

```
mon_arg_list* mon_arg_list_copy          (mon_arg_list *src);
```

Makes a deep copy of a [mon\\_arg\\_list](#).

*src* : a [mon\\_arg\\_list](#).

**Returns** : a deep copy of *src*.

**mon\_arg\_list\_find ()**

```
mon_arg*      mon_arg_list_find          (mon_arg_list *l,  
                                          const char *name);
```

Find a named argument in a [mon\\_arg\\_list](#).

*l* : a [mon\\_arg\\_list](#).

*name* : the name of the argument.

**Returns** : a [mon\\_arg](#) or NULL if it could not be found.

**mon\_arg\_list\_val ()**

```
mon_simple_types* mon_arg_list_val      (mon_arg_list *l,  
                                          const char *name);
```

Like [mon\\_arg\\_list\\_find\(\)](#), but returns the argument's value instead of the [mon\\_arg](#) structure.

*l* : a [mon\\_arg](#) list.

*name* : the name of the argument.

**Returns** : pointer to the argument's value or NULL if it could not be found.

**mon\_cfg\_parse\_file ()**

```
int           mon_cfg_parse_file        (const char *filename,  
                                          mon_cfg_node **cfg);
```

Parse a configuration file.

*filename* : a configuration file name.

*cfg* : on successful return, it contains the root node of the configuration.

**Returns** : 0 or an error code.

**mon\_cfg\_free ()**

```
void          mon_cfg_free              (mon_cfg_node *cfg);
```

Free the configuration tree rooted at *cfg*.

*cfg* : a configuration tree.

**mon\_cfg\_get ()**

```
int          mon_cfg_get          (mon_cfg_node *root,  
                                  const char *key,  
                                  mon_cfg_node **node);
```

Find a node in a configuration tree.  
The *key* parameter has the syntax

```
[section [ '[' sectionname ']' ] "::<" ]* nodename
```

*root* : the root of a configuration tree.

*key* : the name of the node to find.

*node* : on return, it contains the address of the node found.

**Returns** : 0 if successful or a non-0 error code.

**mon\_cfg\_get\_int ()**

```
int          mon_cfg_get_int      (mon_cfg_node *root,  
                                  const char *key,  
                                  int *value);
```

Get the value of an integer configuration parameter.

*root* : the root of a configuration tree.

*key* : the name of the node to find.

*value* : the value of the configuration parameter.

**Returns** : 0 if successful or a non-0 error code.

**mon\_cfg\_get\_bool ()**

```
int          mon_cfg_get_bool     (mon_cfg_node *root,  
                                  const char *key,  
                                  int *value);
```

Get the value of a boolean configuration parameter.

*root* : the root of a configuration tree.

*key* : the name of the node to find.

*value* : the value of the configuration parameter.

**Returns** : 0 if successful or a non-0 error code.

**mon\_cfg\_get\_string ()**

```
int          mon_cfg_get_string          (mon_cfg_node *root,  
                                         const char *key,  
                                         const char **value);
```

Get the value of a string configuration parameter.

*root* : the root of a configuration tree.

*key* : the name of the node to find.

*value* : the value of the configuration parameter.

**Returns** : 0 if successful or a non-0 error code.

**mon\_cfg\_query ()**

```
int          mon_cfg_query              (mon_cfg_node *root,  
                                         const char *key,  
                                         int *type,  
                                         mon_simple_types *val);
```

Get the value of a configuration parameter.

*root* : the root of a configuration tree.

*key* : the name of the node to find.

*type* : on return, it contains the type of the configuration parameter.

*val* : on return, it contains the value of the configuration parameter.

**Returns** : 0 if successful or a non-0 error code.

**mon\_cfg\_first ()**

```
mon_cfg_node* mon_cfg_first            (mon_cfg_node *root,  
                                         const char *name,  
                                         mon_cfg_enum *state);
```

Get the first occurrence of a key in a configuration section.

*root* : a configuration section.

*name* : the name of the node to find. It must be a single component, section selectors are not allowed.

*state* : the internal enumeration state.

**Returns** : the first node found or NULL if *name* does not exist.

### **mon\_cfg\_next ()**

```
mon_cfg_node* mon_cfg_next (mon_cfg_enum *state);
```

Get the next occurrence of a key in a configuration section.

*state* : the internal enumeration state returned by [mon\\_cfg\\_first\(\)](#).

**Returns** : the next node found or NULL if there were no more entries.

### **mon\_cfg\_end ()**

```
void mon_cfg_end (mon_cfg_enum *state);
```

Free memory allocated to a configuration node enumeration.

*state* : the internal enumeration state returned by [mon\\_cfg\\_first\(\)](#).

### **mon\_cfg\_get\_section ()**

```
mon_cfg_node* mon_cfg_get_section (mon_cfg_node *root,  
                                   const char *name,  
                                   const char *title);
```

Get a configuration section.

*root* : the root of a configuration tree.

*name* : the name of the section.

*title* : the title of the section.

**Returns** : a [mon\\_cfg\\_node](#).

### **mon\_ipc\_auth\_client\_init ()**

```
int mon_ipc_auth_client_init (int fd,  
                              void **state);
```

Initialize local authentication on the client side. Returns 0 if no further action needed, MON\_CONTINUE if further steps are necessary, or an error code. If MON\_CONTINUE is returned, *state* contains an opaque pointer containing information about the internal state of authentication.

*fd* : file descriptor to authenticate.

*state* : authentication state.

**Returns** : 0 if successful, MON\_CONTINUE if more steps needed or an error code.

### **mon\_ipc\_auth\_client\_end ()**

```
void mon_ipc_auth_client_end (void *state);
```

Free the internal client authentication state.

*state* : the authentication state.

### **mon\_ipc\_auth\_client\_step ()**

```
int          mon_ipc_auth_client_step      (int fd,  
                                           void *state);
```

Performs the next step of client authentication.

*fd* : the file descriptor to authenticate.

*state* : the internal authentication state.

**Returns** : If 0 is returned, the authentication was successful. If MON\_CONTINUE is returned, the function must be called again. If *fd* is a non-blocking socket, MON\_WANT\_READ or MON\_WANT\_WRITE might be returned indicating that the function must be called again when the file descriptor is available for reading or writing respectively. Any other returned value indicates an authentication failure.

### **mon\_ipc\_auth\_server\_init ()**

```
int          mon_ipc_auth_server_init     (int fd,  
                                           mon_ipc_creds *creds,  
                                           void **state);
```

Initialize local authentication on the server side. Returns 0 if no further action needed, MON\_CONTINUE if further steps are necessary, or an error code. If MON\_CONTINUE is returned, *state* contains an opaque pointer containing information about the internal state of authentication.

*fd* : file descriptor to authenticate.

*creds* : pointer to a [mon\\_ipc\\_creds](#) structure which will hold the peer's credentials if the authentication is successful.

*state* : authentication state.

**Returns** : 0 if successful, MON\_CONTINUE if more steps needed or an error code.

### **mon\_ipc\_auth\_server\_end ()**

```
void         mon_ipc_auth_server_end      (void *state);
```

Free the internal server authentication state.

*state* : the authentication state.

### **mon\_ipc\_auth\_server\_step ()**

```
int          mon_ipc_auth_server_step     (int fd,  
                                           void *state);
```

Performs the next step of client authentication.

*fd* : the file descriptor to authenticate.

*state* : the internal authentication state.

**Returns :** If 0 is returned, the authentication was successful. If `MON_CONTINUE` is returned, the function must be called again. If `fd` is a non-blocking socket, `MON_WANT_READ` or `MON_WANT_WRITE` might be returned indicating that the function must be called again when the file descriptor is available for reading or writing respectively. Any other returned value indicates an authentication failure.

### **mon\_io\_add ()**

```
int          mon_io_add          (int fd,
                                void *ptr);
```

Add a file descriptor to the set of descriptors to be monitored for events.

*fd* : the descriptor to add.

*ptr* : opaque pointer to associate with the descriptor. It will be passed to the `mon_io_callback()` functions.

**Returns :** 0 or an error code.

### **mon\_io\_set\_callback ()**

```
int          mon_io_set_callback (int fd,
                                unsigned event,
                                mon_io_callback cb);
```

Set the callback functions for a file descriptor. You must call `mon_io_add()` with the same `fd` parameter before calling this function.

The `cb` callback function will be called when an event set in the `Param2` mask happens on `fd`.

*fd* : a file descriptor.

*Param2* : the event mask for this callback. Must be the combination of `MON_IO_READ`, `MON_IO_WRITE` and `MON_IO_ERROR`.

*cb* : the callback function.

**Returns :** 0 or an error code.

### **mon\_io\_set\_events ()**

```
int          mon_io_set_events  (int fd,
                                unsigned mask);
```

Add events to the set of watched events. You must call `mon_io_add()` with the same `fd` parameter before calling this function.

*fd* : a file descriptor.

*Param2* : the events to add. Must be the combination of `MON_IO_READ`, `MON_IO_WRITE` and `MON_IO_ERROR`.

**Returns :** 0 or an error code.

**mon\_io\_set\_aux ()**

```
int          mon_io_set_aux          (int fd,  
                                     void *ptr);
```

Modify the opaque pointer associated with the *fd* file descriptor.

*fd* : a file descriptor.

*ptr* : the pointer to associate with the descriptor.

**Returns** : 0 or an error code.

**mon\_io\_clear\_events ()**

```
int          mon_io_clear_events     (int fd,  
                                     unsigned mask);
```

Remove events from the set of watched events.

*fd* : a file descriptor.

*Param2* : the events to remove. Must be the combination of MON\_IO\_READ, MON\_IO\_WRITE and MON\_IO\_ERROR.

**Returns** : 0 or an error code.

**mon\_io\_remove ()**

```
int          mon_io_remove           (int fd);
```

Stop monitoring a file descriptor.

*fd* : a file descriptor.

**Returns** : 0 or an error code.

**mon\_io\_remove\_all ()**

```
void        mon_io_remove_all       (void);
```

Stop monitoring all descriptors. Should only be called when shutting down the library.

**mon\_io\_wait\_events ()**

```
int          mon_io_wait_events      (int timeout,  
                                     int restart);
```

Monitor file descriptors for events and call the appropriate callbacks if necessary.

*timeout* : wait timeout.

*restart* : if true the function will not terminate if the waiting is interrupted by a signal.

**Returns** : 0 or an error code.

## 5 Consumer API

This section describes the consumer library that contains the client API for the monitoring system.

### Description

There are two sets of functions for sending commands to the producer: synchronous and asynchronous.

#### Synchronous

`monp_auth_s()`, `monp_collect_s()`, `monp_stop_s()`, `monp_get_s()`, `monp_subscribe_s()`, `monp_buffer_s()` and `monp_query_s()` send the respective command to the producer and wait for its completion. These functions are expected to operate on blocking-mode handles only. Also, these functions may fail unexpectedly if there is no metric callback installed and an out-of-band metric value arrives before the completion of the command (this is allowed by the protocol). Therefore it is strongly recommended to always install a metric callback using `monp_set_metric_callback()` if these functions are used.

#### Asynchronous

`monp_auth()`, `monp_collect()`, `monp_stop()`, `monp_get()`, `monp_subscribe()`, `monp_buffer()` and `monp_query()` send the respective command to the producer without waiting for its completion. The caller is expected to call `monp_send_async()` to actually send the data, and call `monp_parse_response()` to interpret the results.

### Details

#### `monp_metric_callback ()`

```
void (*monp_metric_callback) (MON *h,
                              mon_metric_value *mv);
```

Specifies the type of the function which is passed to `monp_set_metric_callback()`.

*h* : a **MON** handle where a metric value arrived.

*mv* : a metric value.

#### `monp_cmd_callback ()`

```
void (*monp_cmd_callback) (MON *h,
                           monp_cmd_response *c);
```

Specifies the type of the function which is passed to `monp_set_cmd_callback()`.

*h* : a **MON** handle where a command response arrived.

*c* : a command response.

#### `monp_init ()`

```
void monp_init (const char *config_file);
```

Initialize the consumer library with the given configuration file.

*config\_file* : name of configuration file to use.

### **monp\_done ()**

```
void monp_done (void);
```

Shut down the consumer library and free any resources used internally.

### **monp\_errstr ()**

```
const char* monp_errstr (MON *h,  
int err);
```

Convert a monitoring system error code to a string. This function extends `strerror()`.

*h* : a `MON` handle.

*err* : an error code.

**Returns** : textual description of the error.

### **monp\_module\_init ()**

```
int monp_module_init (mon_cfg_node *cfg);
```

Prototype for loadable consumer module initialization.

*cfg* : the root of the configuration tree for this module (may be NULL).

**Returns** : 0 if the module was successfully initialized. If the returned value is not 0, the module will be unloaded.

### **monp\_module\_done ()**

```
void monp_module_done (void);
```

Prototype for loadable consumer module finalization.

### **monp\_proto\_register ()**

```
int monp_proto_register (monp_proto_module *m);
```

Register a communication protocol module.

*m* : pointer to a `monp_proto_module` to be registered.

**Returns** : 0 on success.

### **monp\_proto\_unregister ()**

```
void monp_proto_unregister (monp_proto_module *m);
```

Unregister a communication protocol module.

*m* : pointer to a `monp_proto_module` to be unregistered.

**monp\_connect ()**

```
MON*          monp_connect          (const char *url);
```

Connect to the monitoring system at the specified *url*.

*url* : an URL to connect to.

**Returns** : a connection handle or NULL on error.

**monp\_listen ()**

```
MON*          monp_listen          (const char *url);
```

Listen on the interface/port specified by *url*.

*url* : an URL to listen on.

**Returns** : a connection handle or NULL on error.

**monp\_accept ()**

```
MON*          monp_accept          (MON *listener);
```

Accept an incoming connection on a listener handle created by [monp\\_listen\(\)](#).

*listener* : a listener handle.

**Returns** : a new handle for the incoming connection or NULL on error.

**monp\_close ()**

```
void          monp_close          (MON *h);
```

Close a connection and release all resources allocated to it.

*h* : a connection handle to close.

**monp\_get\_sd ()**

```
int          monp_get_sd          (MON *h);
```

Get the socket descriptor for the connection handle that can be used in `select`.

*h* : a connection handle.

**Returns** : a socket descriptor.

**monp\_set\_nonblocking ()**

```
int          monp_set_nonblocking          (MON *h,  
                                           int flag);
```

Set the I/O mode of a connection to non-blocking if *flag* is true, or blocking if *flag* is false.

*h* : a connection handle.

*flag* : flag indicating to set or unset non-blocking mode.

**Returns** : 0 or error code.

**monp\_set\_metric\_callback ()**

```
int          monp_set_metric_callback      (MON *h,  
                                           monp_metric_callback cb);
```

Set the callback function to be called when a metric value is received.

*h* : a connection handle.

*cb* : callback function of type [monp\\_metric\\_callback](#).

**Returns** : 0 on success.

**monp\_set\_cmd\_callback ()**

```
int          monp_set_cmd_callback        (MON *h,  
                                           monp_cmd_callback cb);
```

Set the callback function to be called when a command status report is received.

*h* : a connection handle.

*cb* : callback function of type [monp\\_cmd\\_callback](#).

**Returns** : 0 on success.

**monp\_want\_write ()**

```
int          monp_want_write              (MON *h);
```

Check if there is data to be written in the buffer of the connection handle. For example, clients using `select` should set the file descriptor of the handle in the write set if this function returned true.

*h* : a connection handle.

**Returns** : true if there is buffered data to be written, false otherwise.

### **monp\_want\_read ()**

```
int monp_want_read (MON *h);
```

Check if a read operation needs to be performed on the connection handle. For example, clients using `select` should set the file descriptor of the handle in the read set if this function returned true.

*h* : a connection handle.

**Returns** : true if read should be performed, false otherwise.

### **monp\_send ()**

```
int monp_send (MON *h);
```

Send buffered data on a connection handle (blocking).

*h* : a connection handle.

**Returns** : 0 or error code.

### **monp\_send\_async ()**

```
int monp_send_async (MON *h);
```

Send buffered data on a connection handle like `monp_send()` but honour non-blocking mode.

*h* : a connection handle.

**Returns** : 0 or error code.

### **monp\_collect ()**

```
int monp_collect (MON *h,
                  uint32_t *cmd_id,
                  const char *name,
                  mon_arg_list *args);
```

Send a **COLLECT** command to the producer without waiting for completion.

*h* : a connection handle.

*cmd\_id* : a command ID.

*name* : name of the metric to collect.

*args* : a `mon_arg_list` with the metric arguments.

**Returns** : 0 or error code.

### **monp\_stop ()**

```
int          monp_stop          (MON *h,  
                                uint32_t *cmd_id,  
                                uint32_t metric_id,  
                                uint32_t connection);
```

Send a **STOP** command to the producer without waiting for completion.

*h* : a connection handle.

*cmd\_id* : a command ID.

*metric\_id* : a metric ID.

*connection* : a connection ID to stop sending metric values to.

**Returns** : 0 or error code.

### **monp\_get ()**

```
int          monp_get          (MON *h,  
                                uint32_t *cmd_id,  
                                uint32_t metric_id);
```

Send a **GET** command to the producer without waiting for completion.

*h* : a connection handle.

*cmd\_id* : a command ID.

*metric\_id* : a metric ID.

**Returns** : 0 or error code.

### **monp\_subscribe ()**

```
int          monp_subscribe    (MON *h,  
                                uint32_t *cmd_id,  
                                uint32_t metric_id,  
                                uint32_t connection);
```

Send a **SUBSCRIBE** command to the producer without waiting for completion.

*h* : a connection handle.

*cmd\_id* : a command ID.

*metric\_id* : a metric ID.

*connection* : a connection ID to send metric values to.

**Returns** : 0 or error code.

**monp\_buffer ()**

```
int          monp_buffer          (MON *h,
                                   uint32_t *cmd_id,
                                   uint32_t metric_id,
                                   uint32_t connection);
```

Send a **BUFFER** command to the producer without waiting for completion.

*h* : a connection handle.

*cmd\_id* : a command ID.

*metric\_id* : a metric ID.

*connection* : a connection ID to send metric values to.

**Returns** : 0 or error code.

**monp\_auth ()**

```
int          monp_auth          (MON *h,
                                   uint32_t *cmd_id,
                                   const char *name,
                                   const char *authz,
                                   uint32_t credlen,
                                   void *credentials);
```

Send an **AUTH** command to the producer without waiting for completion.

*h* : a connection handle.

*cmd\_id* : a command ID.

*name* :

*authz* :

*credlen* :

*credentials* :

**Returns** : 0 or error code.

**monp\_get\_special ()**

```
int          monp_get_special    (MON *h,
                                   uint32_t *cmd_id,
                                   uint32_t metric_id,
                                   uint32_t param);
```

Send a **GET\_SPECIAL** command to the producer without waiting for completion.

*h* : a connection handle.

*cmd\_id*: a command ID.

*metric\_id*: a metric ID.

*param*:

**Returns** : 0 or error code.

### **monp\_query ()**

```
int          monp_query          (MON *h,
                                uint32_t *cmd_id,
                                const char *name,
                                mon_arg_list *args);
```

Send a **QUERY** command to the producer without waiting for completion.

*h* : a connection handle.

*cmd\_id*: a command ID.

*name* : name of the metric to query.

*args* : a **mon\_arg\_list** with the metric arguments.

**Returns** : 0 or error code.

### **monp\_collect\_s ()**

```
int          monp_collect_s     (MON *h,
                                const char *name,
                                mon_arg_list *args,
                                uint32_t *metric_id);
```

Send a **COLLECT** command to the producer and wait for its completion.

*h* : a connection handle.

*name* : name of the metric to collect.

*args* : a **mon\_arg\_list** with the metric arguments.

*metric\_id* : returned ID for the metric instance.

**Returns** : 0 or error code.

### **monp\_stop\_s ()**

```
int          monp_stop_s       (MON *h,
                                uint32_t metric_id,
                                uint32_t connection);
```

Send a **STOP** command to the producer and wait for its completion.

*h* : a connection handle.

*metric\_id*: a metric ID.

*connection*: a connection ID to stop sending metric values to.

**Returns** : 0 or error code.

#### **monp\_get\_s ()**

```
int          monp_get_s                (MON *h,  
                                       uint32_t metric_id);
```

Send a **GET** command to the producer and wait for its completion.

*h*: a connection handle.

*metric\_id*: a metric ID.

**Returns** : 0 or error code.

#### **monp\_subscribe\_s ()**

```
int          monp_subscribe_s          (MON *h,  
                                       uint32_t metric_id,  
                                       uint32_t connection);
```

Send a **SUBSCRIBE** command to the producer and wait for its completion.

*h*: a connection handle.

*metric\_id*: a metric ID.

*connection*: a connection ID to send metric values to.

**Returns** : 0 or error code.

#### **monp\_buffer\_s ()**

```
int          monp_buffer_s             (MON *h,  
                                       uint32_t metric_id,  
                                       uint32_t connection);
```

Send a **BUFFER** command to the producer and wait for its completion.

*h*: a connection handle.

*metric\_id*: a metric ID.

*connection*: a connection ID to send metric values to.

**Returns** : 0 or error code.

### **monp\_auth\_s ()**

```
int          monp_auth_s          (MON *h,  
                                  const char *name,  
                                  const char *authz,  
                                  uint32_t credlen,  
                                  void *credentials,  
                                  uint32_t *channel_id);
```

Send an **AUTH** command to the producer and wait for its completion.

*h* : a connection handle.

*name* :

*authz* :

*credlen* :

*credentials* :

*channel\_id* :

**Returns** : 0 or error code.

### **monp\_get\_special\_s ()**

```
int          monp_get_special_s   (MON *h,  
                                  uint32_t metric_id,  
                                  uint32_t param);
```

Send a **GET\_SPECIAL** command to the producer and wait for its completion.

*h* : a connection handle.

*metric\_id* : a metric ID.

*param* :

**Returns** : 0 or error code.

### **monp\_query\_s ()**

```
int          monp_query_s        (MON *h,  
                                  const char *name,  
                                  mon_arg_list *args,  
                                  uint32_t *metric_id);
```

Send a **QUERY** command to the producer and wait for its completion.

*h* : a connection handle.

*name* : name of the metric to query.

*args* : a **mon\_arg\_list** with the metric arguments.

*metric\_id* : returned ID for the metric instance.

**Returns** : 0 or error code.

### **monp\_parse\_response ()**

```
int          monp_parse_response          (MON *h,  
                                          uint32_t *metric_id,  
                                          monp_response *r);
```

Try to read and parse data from the producer. If metric and/or command callbacks are registered, they will be called to handle the returned data and this function will always return `MON_CONTINUE`. If one or both of the callbacks are not registered, the data will be returned directly in *metric\_id* and *r*.

*h* : a connection handle.

*metric\_id* : returned metric ID.

*r* : returned `mon_response` structure.

**Returns** : 0 on success, `MON_CONTINUE` if there is more data available or error code.

### **monp\_wait\_metric ()**

```
int          monp_wait_metric            (MON *h,  
                                          uint32_t metric_id,  
                                          mon_metric_value **mv);
```

Wait until a value for the metric identified by *metric\_id* arrives and return it in *mv*.

*h* : a connection handle.

*metric\_id* : a metric ID.

*mv* : returned `mon_metric_value` structure.

**Returns** : 0 or error code.

### **monp\_metric\_type ()**

```
mon_type*    monp_metric_type           (MON *h,  
                                          uint32_t metric_id);
```

Get a `mon_type` structure describing the type of the metric identified by *metric\_id*.

*h* : a connection handle.

*metric\_id* : a metric ID.

**Returns** : a `mon_type` structure or `NULL` on error.

## 6 Producer API

This section documents the producer API used by the producer daemons and sensor modules.

### 6.1 Producer loadable module support

#### Description

Module API

#### Details

##### **prod\_metric\_instance\_new ()**

```
prod_metric_instance* prod_metric_instance_new
                                (mon_metric_def *md,
                                mon_arg_list *args);
```

Create a new metric instance using *md* as the metric definition and *args* as the values of the formal parameters.

*md* : a [mon\\_metric\\_def](#).

*args* : a [mon\\_arg\\_list](#).

**Returns** : a [prod\\_metric\\_instance](#) or NULL on error.

##### **prod\_metric\_instance\_free ()**

```
void prod_metric_instance_free (prod_metric_instance *mi);
```

Free a metric instance.

*mi* : a [prod\\_metric\\_instance](#).

##### **sensor\_register ()**

```
int sensor_register (void *mod_handle,
                    prod_sensor_module *m);
```

Register a sensor.

*mod\_handle* : the handle for the module registering the sensor.

*m* : a [prod\\_sensor\\_module](#).

**Returns** : 0 or an error code.

##### **sensor\_unregister ()**

```
void sensor_unregister (prod_sensor_module *m);
```

Unregister a sensor.

*m* : a [prod\\_sensor\\_module](#).

### **sensor\_send ()**

```
void sensor_send (prod_metric_value *mv);
```

Send a metric value generated by a metric instance. The producer library will route the metric value to every channels subscribed to that metric instance.

*mv* : a **prod\_metric\_value**.

### **sensor\_check\_metric ()**

```
int sensor_check_metric (const sensor_simple_desc *desc,  
mon_metric_def *md);
```

Helper function for simple sensor modules implementing the **check\_metric()** method of **prod\_sensor\_module**.

*desc* : description table containing the metrics served by the sensor.

*md* : the requested metric definition.

**Returns** : 0 if *md* matches an element in *desc*, an error code otherwise.

### **sensor\_start\_instance ()**

```
prod_metric_instance* sensor_start_instance (const sensor_simple_desc *desc,  
mon_metric_def *md,  
mon_arg_list *args);
```

Helper function for simple sensor modules implementing the **start\_instance()** method of **prod\_sensor\_module**.

*desc* : description table containing the metrics served by the sensor.

*md* : the requested metric definition.

*args* : arguments of the metric instance.

**Returns** : a **prod\_metric\_instance** or NULL on error.

### **prod\_proto\_register ()**

```
int prod_proto_register (void *mod_handle,  
prod_proto_module *m);
```

Registers a protocol handler.

*mod\_handle* : the handle for the module registering the protocol.

*m* : a **prod\_proto\_module**.

**Returns** : 0 or an error code.

### **prod\_proto\_unregister ()**

```
void      prod_proto_unregister      (prod_proto_module *m);
```

Unregister a protocol handler.

*m* : a **prod\_proto\_module**.

### **prod\_module\_init ()**

```
int      prod_module_init      (void *mod_handle,  
                                mon_cfg_node *cfg);
```

Prototype for loadable producer module initialization.

*mod\_handle* : the handle for the module.

*cfg* : the root of the configuration tree for this module (may be NULL).

**Returns** : 0 if the module was successfully initialized. If the returned value is not 0, the module will be unloaded.

### **prod\_module\_done ()**

```
void      prod_module_done      (void);
```

Prototype for loadable producer module finalization.

## **6.2 Producer library**

### **Description**

Producer library

### **Details**

#### **prod\_lib\_init ()**

```
int      prod_lib_init      (const char *config_file);
```

Initialize the producer library.

*config\_file* : the configuration file to use.

**Returns** : 0 or an error code.

#### **prod\_lib\_done ()**

```
void      prod_lib_done      (void);
```

Shut down the producer library and free allocated resources.

**prod\_log\_open ()**

```
void      prod_log_open      (void);
```

(Re)open the logfile.

**prod\_log\_target ()**

```
int      prod_log_target      (const char *target);
```

Set the logging target. The *target* parameter can have the following values:

*target* : the log target.

**Returns** : 0 or an error code.

**prod\_log\_ident ()**

```
int      prod_log_ident      (const char *ident);
```

Set the identity string used in log messages.

*ident* : the identity string.

**Returns** : 0 or an error code.

**prod\_log\_level ()**

```
int      prod_log_level      (int level);
```

Set the log level. You can use the same levels as for syslog.

*level* : the log level.

**Returns** : 0 or an error code.

## 6.3 Common producer functions

### Description

Producer functions that modules are allowed to use too

## Details

### **prod\_metric\_value\_new ()**

```
prod_metric_value* prod_metric_value_new      (prod_metric_instance *mi,  
                                              prod_metric_id *mid);
```

Allocates a new **prod\_metric\_value**.

The *mi* parameter specifies the metric instance this value belongs to. When the value should go only to one channel (e.g. as a result of a GET command), *mid* specifies the **prod\_metric\_id** where it should go. If the value should go to every subscribed channels, *mid* is NULL.

*mi* : the **prod\_metric\_instance** this value belongs to.

*mid* : the **prod\_metric\_id** this value belongs to.

**Returns** : 0 or an error code.

### **prod\_metric\_value\_conv ()**

```
prod_metric_value* prod_metric_value_conv    (prod_metric_instance *mi,  
                                              prod_metric_id *mid,  
                                              mon_metric_value *smv);
```

Convert a **mon\_metric\_value** to a **prod\_metric\_value**.

*mi* : the **prod\_metric\_instance** this value belongs to.

*mid* : the **prod\_metric\_id** this value belongs to.

*smv* : the original **mon\_metric\_value**.

**Returns** : 0 or an error code.

### **prod\_metric\_value\_done ()**

```
void          prod_metric_value_done        (prod_metric_value *mv);
```

Drop a reference to a **prod\_metric\_value**. When the last reference is gone, the memory allocated to *mv* will be freed.

*mv* : a **prod\_metric\_value**.

### **prod\_sched\_once ()**

```
int          prod_sched_once                (prod_sched_callback cb,  
                                              void *data,  
                                              time_t when);
```

Schedule a function to be called at a specific time.

*cb* : a callback function.

*data* : pointer to be passed to the callback function.

*when* : time when *cb* should be called.

**Returns** : 0 or an error code.

**prod\_sched\_periodic ()**

```
int          prod_sched_periodic          (prod_sched_callback cb,  
                                          void *data,  
                                          time_t freq);
```

Periodically call a function.

*cb* : a callback function.

*data* : pointer to be passed to the callback function.

*freq* : frequency to call *cb*.

**Returns** : 0 or an error code.

**prod\_sched\_remove ()**

```
int          prod_sched_remove          (prod_sched_callback cb,  
                                         void *data);
```

Remove a scheduled callback.

*cb* : a callback function.

*data* : pointer to be passed to the callback function.

**Returns** : 0 or an error code.

**prod\_sched\_freq\_mod ()**

```
int          prod_sched_freq_mod        (prod_sched_callback cb,  
                                          void *data,  
                                          time_t freq);
```

Modify the calling frequency of a function scheduled for periodic calling.

*cb* : a callback function.

*data* : pointer to be passed to the callback function.

*freq* : the new frequency.

**Returns** : 0 or an error code.

**prod\_errstr ()**

```
const char* prod_errstr                (int error);
```

Convert a monitoring system error code to a string. This function extends `strerror()`.

*error* : an error code.

**Returns** : textual description of the error.

## 6.4 Producer data structures

Producer data structures – Producer data structures

### Description

Producer data structures

### Details

#### **prod\_sched\_callback ()**

```
void (*prod_sched_callback) (void *ptr);
```

Specifies the type of the function which is passed to `prod_sched_once()` or `prod_sched_periodic()`.

*ptr* : an opaque pointer.