Measurement Handler Application (MESH)

version 1.1

Fredrik Gustafson

1998-04-24

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

Measurement Handler (MESH) User's Guide

The Measurement Handler Application (MESH) is an Operation and Maintenance application. It contains support for applications to register measurement types and create measurement objects, which in turn perform measurements and report them to MESH.

MESH handles common tasks, such as threshold checking and some of the supervision. MESH also provides manager functions for logging measurement reports and controlling the measurement types and objects.

MESH is a management protocol independent application that needs protocol adaptations to communicate with a remote manager. Currently, an SNMP adaptation is included in the MESH application.

1.1 Introduction

The operation and maintenance support in OTP consists of a generic model for management subsystems in OTP and some components to be used in these subsystems. The model that this support is based upon, is described in “OAM Principles”.

This document describes one of these components, the Measurement Handler application MESH. MESH consists of functions, which support, create and control measurement types and objects, and associated logs.

MESH uses the applications SASL, Mnesia, and EVA.

Architecture

MESH is designed to work as a distributed application, which means that it always executes on one node, but will be restarted on a standby node if the current node goes down. MESH should run on the same node as other operation and maintenance applications, specifically the management protocol termination application, to minimize internal network traffic.

MESH is designed to be protocol independent, and may be used with different management protocols. For each such protocol, a *MESH adaptation* must be written. Currently, an SNMP adaptation is included, and other adaptations (e.g., HTTP, CORBA) may follow in the future.

The Measurement Handler system can run in both *server* mode and in *client* mode. In client mode, no processes are running, but the code implementing the API is loaded. There must always be one MESH server running on a node in a network of Erlang nodes.

For MESH, all involved nodes are seen as one (distributed) system. This means for example that there is one active alarm list (in EVA) for the entire system.

1.2 Services

The purpose of this document is to give an understanding of how the Measurement Handler works, and how to build applications to interact with MESH. The intention is *not* to give all details about the functions making up the MESH API. For a complete description of this API, please see the `mesh` Reference Manual.

For a description of the statistical functions that are available through the Measurement Handler application, please see the `mesh_lib` Reference Manual.

The main purpose of the Measurement Handler is to provide the user with a simplified and uniform way of handling measurement data, as well as managing the applications and processes generating the data. To accomplish this, the Measurement Handler application takes care of common tasks, such as creation and control of measurement applications/processes, node and measurement application supervision, threshold checking and notification, tide-mark handling, and so on. A model is available to the user for implementing the applications and processes performing the actual measurements.

Services Offered by the Measurement Handler

The following is a complete list of the services the Measurement handler currently offers:

- measurement type registration
- book-keeping of measurement types
- administration of measurement types (i.e., locking, unlocking and shutting down)
- supervision of measurement types
- control of the total number of measurement types (reporting to EVA)
- creation of measurement objects
- book-keeping of measurement objects
- administration of measurement object (i.e., starting and stopping them)
- resetting of measurement objects
- supervision of measurement objects (at least part of the supervision scheme
- revival of disabled measurement objects, using the last known settings
- control of the total number of measurement objects (reporting to EVA)
- control of the number of measurement objects belonging to the same measurement type (reporting to EVA)
- supervision of nodes used by measurement types and objects
- storage and logging of measurement reports received from measurement objects
- threshold support for measurement objects
- threshold administration (i.e., enabling and disabling specified thresholds)
- tide-mark support for measurement objects
- logging of events and alarms generated
- support for statistical handling of measurement samples
- implementation models for measurement types and objects

Basic Concepts

A *Measurement Object* is a user provided application or process collecting and filtering measurement samples provided by one or more resource objects, see below. Measurement objects are grouped together in Measurement Types.

A *Measurement Type* may be regarded as a container, holding all the code needed when dealing with the associated measurement objects. Specifically, the measurement type shall provide an interface, which is used by the Measurement Handler to create, delete, and control the measurement objects.

A *Resource Object* is a user or operating system provided application or device producing measurement samples.

Measurement types and objects exhibit three groups of states that are supported by the Measurement Handler:

- *operability states*: whether or not the type/object is physically installed and working.
  A measurement object may be in one of the following states:
  - `enabled`: it has been created and is working as intended.
  - `disabled`: it has been created but no longer works, e.g., due to a crash.

  A measurement type may only be in an `enabled` state.
- *usage states*: whether or not the type/object is actively in use, and if so, whether or not it has spare capacity for additional users.
  A measurement type may be in one of the states below:
  - `idle`: no measurement objects have been created.
  - `active`: some measurement objects have been created, but one or more objects may still be created.
  - `busy`: the maximum number of measurement objects allowed (set during measurement type registration) has been reached.

  A measurement object may only be in a `busy` state.
- *administration states*: permission to use or prohibition against using the type/object, imposed through the management service.
  A measurement type may be in one of the following states:
  - `locked`: the type can currently not be used. Existing measurement objects will immediately be deleted.
  - `shutting down`: use of the type is permitted to already existing measurement objects only. Once all objects have been deleted, the type will automatically enter state `locked`.
  - `unlocked`: no restrictions. new measurement objects may freely be created (provided the usage state doesn't hinder this).

  A measurement object may be in one of the following states:
  - `stopped`: the object is idle.
  - `started`: the object is running, performing its duties.

Implementation of Measurement Objects

Basically a measurement object consists of a server, polling one or more resource objects for measurement samples, which are filtered and reported to the Measurement Handler. However,if supervision is desired, parts of the supervision scheme have to be implemented by the user. This leads us to the concept of Measurement Responsible Processes, MRPs.

A *Measurement Responsible Process* may be regarded as the gateway between the measurement type interface and the individual measurement objects (belonging to that specific measurement type). There may be many measurement objects, but we want to access all of them with the same function in the measurement type interface, meaning that we have to provide the (state-less) interface with information about how to contact a certain measurement object. There are basically two ways of doing this:

- we provide the interface with the address (the process identifier) to each measurement object we are interested in. In this case we say that the measurement object is its own MRP.
- we provide the measurement interface with the address (the process identifier) to a server, which in turn keeps track of the existing measurement objects and forwards the order/request to the correct measurement object. Using this design we may address the measurement objects using logical names, letting the server translate the name to a process identifier. In this case the server is the MRP.

The above alternatives are a matter of choice, depending on the supervision required; this is explained in greater detail below. MESH supports both alternatives, provided the user follows some basic rules. The first one is to ensure that the MRP fulfills the following four requirements:

- it shall supervise one or more measurement objects.
- it shall map the measurement object identifier to the correct process identifier, thereby enabling communication with measurement objects.
- it shall keep track of the available resources, mapping new measurement objects to the correct resource.
- it shall by its mere existence ensure MESH that the user supplied supervision scheme is still working. That is, MESH only supervises the MRP (and the node it resides on); the MRP has to supervise every individual measurement object, reporting to MESH whenever one gets disabled (and order revival, if that is desired).
  Note: should the MRP not report measurement object termination to MESH, the operation of the Measurement Handler is by no means guaranteed - most probably it will fail!!!

Secondly, should an MRP terminate, *MESH assumes that the corresponding measurement objects also terminates!* In short, an MRP has to be very robust, i.e., it has to trap exits, even internally (for example by letting the inner loop be surrounded by a catch-statement, where run-time errors may be caught and the inner loop re-entered).

Finally, when MESH calls functions in the measurement type interface, the MRP process identifier should be supplied as one of the arguments. The measurement type interface forwards the order to the MRP, e.g., by sending a message to the MRP, which in turn performs the order, or forwards it to the correct measurement object. For this to work, it is required that the measurement type interface (when the creation of a new measurement object is ordered), returns the process identifier to the MRP responsible for the new object.

Three suggested MRP models are described extensively below, in section “MRP Models”.

MRP Models

As seen previously, in the section “Implementation of Measurement Objects”, it is required that all measurement objects are linked to a Measurement Responsible Process. The Measurement Responsible Process must also fulfill some requirements.

There are three main implementation cases:

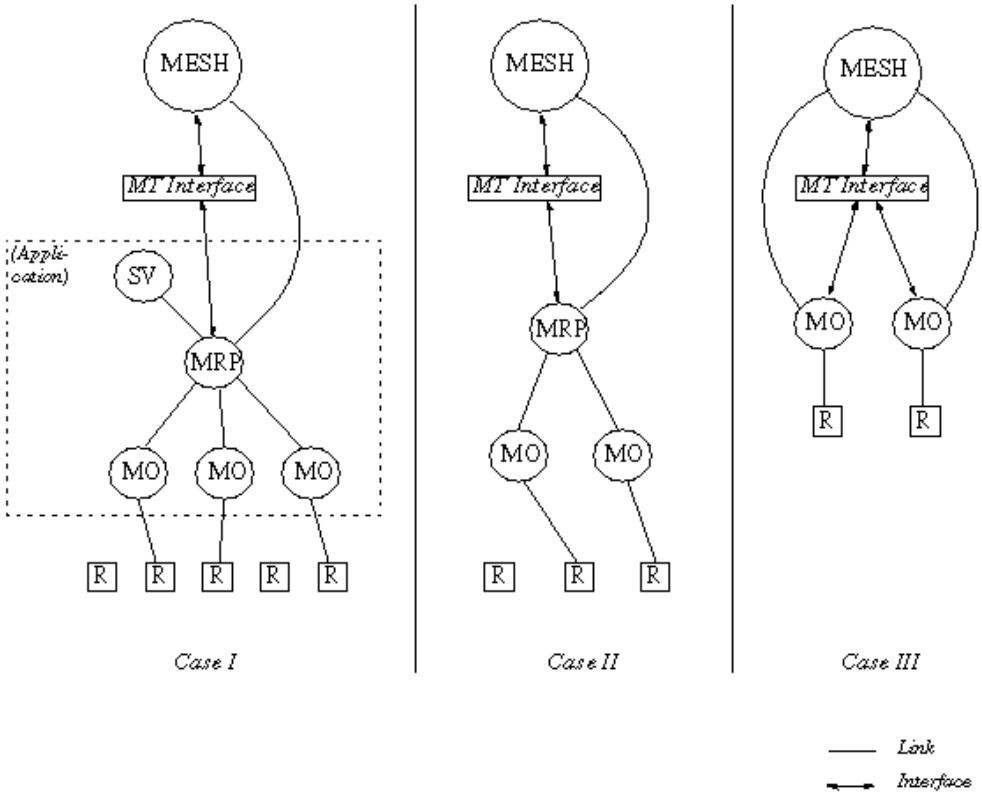

Figure 1.1: MRP Models. (R=Resource Object, SV=Supervisor, MO=Measurement Object)

Each of the three cases is described in depth below.

Case I - applications and OTP supervision

In this case the measurement objects are part of an application. The OTP mechanisms for supervision are fully used, and the OTP mechanisms for changing code in runtime *may* be used.

When the application is started, it registers the corresponding measurement type in MESH. This causes MESH to call the `init` function in the measurement type interface; this function will return (to MESH) the Process Identifier of the MRP (it may even start the MRP).

Note:
A registered name is not adequate, since MESH uses the PID to get the node where the MRP resides!

Now that the measurement type has been registered, measurement objects may be created. This causes MESH to call the `create_measurement` function in the measurement type interface. It is required that this function returns the process identifier of the associated MRP (the one mentioned above), which MESH will supervise. MESH will also supervise the node where the MRP is residing.

When MESH wants to communicate with any measurement object, using the measurement type interface, it is the responsibility of the MRP to map the measurement object identifier to the process identifier in question, and to forward the order/message to that process. (The MRP may for example internally keep a list of tuples {`MeasurementID, PID`}.

Should a measurement object terminate, the MRP will notify MESH (and also decide if it should request revival of the object).

Should the MRP itself terminate, e.g., due to a node crash, MESH will use EVA to notify the manager about the disabled measurement objects (or unconnected, depending on the type of crash). MESH will then wait for the node to be restarted. If just the connection to the node was lost MESH will try to re-connect to the MRP. If the attempt is successful, the manager will be informed (via EVA) about the measurement type and the measurement objects being reconnected, otherwise MESH will wait for the application to be restarted, the type re-registered and the measurement objects revived.

Should the application restart, it will once again try to register the measurement type, whereupon it will be informed (using the return value of the registration function) by MESH that the type has already been registered. The application may now choose to ask MESH to restart the measurement objects it is aware of (i.e., those measurement objects that haven't been explicitly deleted by the manager or the application). It shall be noted that the registration attempt will cause MESH to once again call the `init` function in the measurement interface, since it has to get the process identifier of the new MRP!

Case II - no applications, and simplified supervision

In this case the measurement objects are not part of an application, and the OTP mechanisms for supervision are not fully utilized. Instead, the MRP supervises the measurement objects it has started, restarting them (or requesting MESH to restart them) when necessary. MRP itself is not supervised by any other process than MESH. MESH, in turn, only notifies the manager if the MRP terminates. Therefore, the MRP *is not* restarted unless the manager explicitly orders MESH to re-register the measurement type (more about this below).

A process, or the manager, registers the measurement type in MESH. As previous, this will cause MESH to call the `init` function in the measurement type interface; this function will return the process identifier of the MRP.

Now that the measurement type has been registered, measurement objects may be created. MESH will then call the `create_measurement` function in the measurement type interface. This function returns the process identifier of the associated MRP (the one mentioned above), which MESH will supervise. MESH will also supervise the node where the MRP is residing.

As in the previous case, the MRP will handle the mapping of measurement object identifiers to the correct corresponding process identifier, forwarding messages and orders to the correct recipient. Should a measurement object terminate, MRP shall notify MESH, and decide if MESH should revive the measurement object.

Should the MRP itself terminate, for example, due to a node crash, MESH will use EVA to notify the manager about the corresponding measurement objects being disabled (or unconnected, depending on the type of crash). MESH will then wait for the node to be started again. In instances where only the

connection to the node is lost MESH will try to re-connect to the MRP. Should the attempt succeed, the manager will (via EVA) be informed about the measurement type and the measurement objects being reconnected. Otherwise MESH will wait for the application to be restarted, the measurement type re-registered and the measurement objects revived.

Note:
Should the manager directly order revival of the measurement objects (`mesh:revive_measurement`), the attempt will fail and the return value will inform the manager the cause of the previous termination of the MRP.

When re-registration has been ordered and completed, MESH may be ordered to revive the objects it is aware of (i.e., those measurement objects that haven't been explicitly deleted by the manager or the application).

Note:
Even in this case the registration attempt will again causes MESH to call the `init` function in the measurement interface, since it has to get the process identifier of the new MRP!

It should be noted that the OTP mechanisms for changing code in runtime cannot be used in this case.

Case III - each measurement object is its own MRP

In this case each measurement object also functions as an MRP. This means that if the measurement objects are terminated they won't be restarted, unless MESH is explicitly ordered by the manager to revive them.

A process, or the manager, will register the measurement type in MESH. This causes MESH to call the `init` function in the measurement type interface; unlike Case I and II, this function will return the atom `undefined` (since there is no common MRP here).

Now measurement objects may be created. As above, this results in the `create_measurement` function in the measurement type interface being called. It is required that this function returns the MRP PID associated with the measurement object, i.e., the PID of the measurement object itself! MESH will supervise this process, and also the node where it's residing.

Should a measurement object terminate, MESH will, using EVA, notify the manager of this, whereupon MESH will wait for further orders. Provided the node in question is up, the manager may order MESH to revive the measurement objects. Once again, when the `create_measurement` function in the measurement type interface is called, and will return the PID of the MRP/measurement object.

Should it be unclear whether a measurement object/MRP has actually terminated (for example, if the connection to a node has been lost), MRP will wait for the node to be connected once again. MESH will then check whether or not the measurement object/MRP still exists. If it does, the manager will be informed (via EVA) that it still exists or that the the measurement object has been disabled.

It should be noted that the OTP mechanisms for changing code in runtime cannot be used in this case.

Working with the Measurement Handler

In the following example we will take a close look at how to start and work with the Measurement Handler. This example demonstrates the functionality of the Measurement Handler, it is run on only one node and is extremely simplified and therefore may not show the optimal way to configure the system and the different applications required.

We will be using the simple interface module below, `diversity.erl`. This module uses the “Case II” model described in section “MRP Models”, as described in the previous chapter.

```
-module(diversity).

%% Necessary type interface functions!
-export([init/1,
         terminate/1,
         create_measurement/5,
         delete_measurement/3,
         start_measurement/3,
         stop_measurement/2,
         reset_measurement/3]).

-export([mrp/0,
         meas/2]).


%% A simple measurement type interface.

init(TypeName) ->
    spawn(?MODULE, mrp, []).

terminate(undefined) ->
    ok;
terminate(MRP) ->
    MRP ! terminate,
    ok.

create_measurement(undefined, _Type, _Meas, _Res, _Args) ->
    {error, no_mrp};
create_measurement(MrpInit, TypeName, MeasName, ResId, InitArgs) ->
    MrpInit ! {create_measurement, self(), MeasName, InitArgs},
    receive
        {Pid,MrpInit} ->
            MrpInit
    end.

start_measurement(MRP, MeasName, Args) ->
    MRP ! {start, MeasName, Args},
    ok.

stop_measurement(MRP, MeasName) ->
    MRP ! {stop, MeasName},
    ok.
```

```
reset_measurement(MRP, MeasName, Args) ->
    MRP ! {reset, MeasName, Args},
    ok.

delete_measurement(MRP, MeasName, Args) ->
    MRP ! {delete_measurement, MeasName, Args},
    ok.


%% The code implementing a Measurement Responsible Process.

mrp() ->
    process_flag(trap_exit, true),
    catch mrp_loop([]),
    mrp().

mrp_loop(MeasList) ->
    receive
        {create_measurement, Sender, MeasId, InitArgs} ->
            Pid = spawn_link(?MODULE, meas, [self(), MeasId]),
            Sender ! {Pid, self()},
            mrp_loop([{MeasId, Pid} | MeasList]);
        {delete_measurement, MeasId, InitArgs} ->
            case lists:keysearch(MeasId, 1, MeasList) of
                false ->
                    done;
                {value, {MeasId,Pid}} ->
                    exit(Pid, kill)
            end,
               % We remove the measurement from the list when the
               % exit signal is received.
            mrp_loop(MeasList);
        {start, MeasId, Args} ->
            send_to_meas(MeasId, MeasList, {start,Args}),
            mrp_loop(MeasList);
        {stop, MeasId} ->
            send_to_meas(MeasId, MeasList, stop),
            mrp_loop(MeasList);
        {reset, MeasId, Args} ->
            send_to_meas(MeasId, MeasList, {reset,Args}),
            mrp_loop(MeasList);
        {'EXIT', Pid, Reason} ->
            case lists:keysearch(Pid, 2, MeasList) of
                false ->
                    mrp_loop(MeasList);
                {value, {MeasId,Pid}} ->
                    mesh:measurement_terminated(MeasId, Reason),
                    mrp_loop(lists:keydelete(Pid, 2, MeasList))
            end;
        Other ->
            mrp_loop(MeasList)
    end.
```

```
send_to_meas(MeasName, MeasList, Msg) ->
    case lists:keysearch(MeasName, 1, MeasList) of
        false ->
            done;
        {value,{Meas,Pid}} ->
            Pid ! Msg
    end.


%% The code implementing a measurement object.

meas(MRP, Name) ->
    process_flag(trap_exit, true),
    meas_loop(MRP, Name, []).

meas_loop(MRP, Name, Args) ->
       % A simple measurement object loop where we ignore
       % start, stop and reset orders!
    receive
        {start,NewArgs} ->
            meas_loop(MRP, Name, NewArgs);
        stop ->
            meas_loop(MRP, Name, Args);
        {reset,NewArgs} ->
            meas_loop(MRP, Name, NewArgs);
        {'EXIT', MRP, Reason} ->
            done;
        _Other ->
            meas_loop(MRP, Name, Args)
    end.
```

Preparation and Start of MESH

In order to start MESH the applications MESH requires must first be started:

- SASL
- Mnesia
- EVA

First we make sure SASL is started:

```
1> application:start(sasl).
ok
```

Next, we prepare for and start Mnesia (running on a single node only). Mnesia is used by MESH to persistently store data about measurement types and measurement objects, see chapter _____ for further information.
First of all, a schema must be created on each node where the MESH application may run (in this example - one node).

```
2> mnesia:create_schema([]).
ok

3> application:start(mnesia).
ok
```

We now proceed by preparing for and starting EVA, which is used by MESH to send and log events and alarms. It should be noted that MESH doesn't supervise EVA, meaning that normally this has to be taken care of by the client application programmer.
It should also be noted that in the most cases, when EVA runs on the same node as MESH internal communication is reduced improving performance.
The first step in the EVA preparations consists of creating the Mnesia tables needed by the basic EVA service and the EVA log service, on each of the nodes where the MESH application is supposed to be running.

```
4> eva_sup:create_tables_log([]).
ok

5> disk_log:open([{name,"default_log"},{format,internal},{type,wrap},{size,{10000,10}}]).
{ok,"default_log"}

6> eva_sup:start_link_log("default_log").
{ok,<0.153.0>}

7> application:start(eva).
ok
```

NOTE: If we intend to use the MESH SNMP adaptation, the previous commands shall instead be:

```
4> eva_sup:create_tables_log_snmp([]).
ok

5> disk_log:open([{name,"default_log"},{format,internal},{type,wrap},{size,{10000,10}}]).
{ok,"default_log"}

6> eva_sup:start_link_log_snmp("default_log", LogDir, MaxDirSize)
{ok,<0.153.0>}

7> application:start(eva).
ok
```

We may now finally start MESH. Firstly, we must create the Mnesia tables needed by MESH.
(Currently two tables are used by MESH, `mesh_type` and `mesh_meas`. They are replicated to disk and RAM on each node that may run the MESH application.)

```
8> mesh:create_tables([]).
ok

9> application:start(mesh).
ok
```

Note:
if we want the MESH SNMP adaptation to be started, the environment variable `snmp_adapted` must be set to `true`!

Registering Measurement Types

Now all preparations are finished, and we can start registering measurement types. We begin by registering a type called *diversity1*, using the `diversity` interface module. Specify the number of measurement objects, belonging to this type, that may be created (up to ten). The argument containing any optional description of the type should be set to the string “A simple measurement type”.

```
10> mesh:register_type(diversity1,"A simple measurement type",diversity,5).
{registered,diversity1}
```

We register one more type, from which it may be possible to create 10 measurement objects:

```
11> mesh:register_type(diversity2,"Another measurement type",diversity,10).
{registered,diversity2}
```

When we register a type, MESH will call the `init` function in the interface module. In our example, this function spawns an MRP process, and returns the process identifier. MESH links itself to the new MRP, and stores the PID for further usage - it will be used whenever operations are performed on, or associated with, the measurement type.

We may ask the Measurement handler for types currently registered:

```
12> mesh:list_types().
[{diversity1,[{extra,"A simple measurement type"},
              {interface_mod,diversity},
              {instances,0},
              {max_instances,5},
              {administrative_state,unlocked}]},
 {diversity2,[{extra,"Another measurement type"},
              {interface_mod,diversity},
              {instances,0},
              {max_instances,10},
              {administrative_state,unlocked}]}]
```

In the listing we see that the administrative state of both types is `unlocked`, and that the number of instances, i.e., the number of measurement objects created, is zero.

Creating Measurement Objects

Now that we have registered measurement types, we may start creating measurement objects. We call this first measurement object *div11*, specifying it belongs to the type *diversity1*. Since we don't have any resource objects in this example, we may let the resource argument contain a zero. The argument containing any optional description of the measurement object should be set to the string "A simple measurement object". We set the original administrative state to `started`, and specify the initial arguments to be the list `[1,2,3]`.

```
13> mesh:create_measurement(div11,diversity1,"A simple measurement object",0,started,[1,2,3]).
{created,div11}
```

When we create a measurement object, MESH will call the `create_measurement` function in the interface module. However, in this example the two measurement types *diversity1* and *diversity2* share the same interface module.

> Note:
> It is possible to have unique types sharing the same module.

As shown above each type has a corresponding MRP - and when MESH calls the `create_measurement` function, one of the arguments is the process identifier of the correct MRP. The interface forwards the create order to this MRP, which in turn spawns a process implementing the measurement object *div11*. When the type interface finally returns, the return value shall contain the process identifier of the process (ie. Measurement Responsible Process for *div11*). From now on, when managing *div11*, every function call MESH does will contain the MRP PID returned when *div11* was created. This is illustrated in the code above.

In Cases I & II the MRP PID must return the same value as when registering the type. In Case III a PID value will be returned which cannot be the same as the returned value (undefined) when registering the type. In all cases, the only requirement is that the MRP fulfills the responsibilities that were listed previously
MESH will ensure that it is linked to the MRP, and store the PID for further usage. As mentioned above, it will be used whenever we want to use the measurement object.

At any time it is possible to ask the Measurement Handler for a list of the measurement objects created:

```
17> mesh:list_measurements(diversity1).
[{div11,[{extra,"A simple measurement object"},
         {resources,0},
         {initial_arguments,[1,2,3]},
         {operability_state,enabled},
         {administrative_state,started}]}]
```

Measurement Type and Object Administration

This section outlines how the registered measurement types and the created measurement objects may be controlled.

Measurement objects may be *started*, *stopped* and *reset*. Each time we start or reset a measurement object, we may specify a new list of arguments, manipulating the internal state of the measurement object. It is important that these argument lists have the same format as the argument list specified when the measurement object was created. This is due to the revival procedure, which will use the last known argument list when re-creating the measurement object; we will take a closer look at this later on in this document.

```
18> mesh:stop_measurement(div11).
{stopped,div11}

19> mesh:list_measurements(diversity1).
[{div11,[{extra,"A simple measurement object"},
         {resources,0},
         {initial_arguments,[1,2,3]},
         {operability_state,enabled},
         {administrative_state,stopped}]}]

20> mesh:start_measurement(div11, [4,5,6]).
{started,div11}

21> mesh:list_measurements(diversity1).
[{div11,[{extra,"A simple measurement object"},
         {resources,0},
         {initial_arguments,[4,5,6]},
         {operability_state,enabled},
         {administrative_state,started}]}]

22> mesh:reset_measurement(div11,[7,8,9]).
{reset,div11}

23> mesh:list_measurements(diversity1).
[{div11,[{extra,"A simple measurement object"},
         {resources,0},
         {initial_arguments,[7,8,9]},
         {operability_state,enabled},
         {administrative_state,started}]}]
```

For each of these orders, MESH calls the corresponding function in the measurement type interface, with one argument being the PID of the MRP associated with measurement object *div11*. It should be noted, that it is the measurement type interface, via the MRP, that actually performs the orders, while MESH in this case has a book-keeping functionality.
In the list of functions above, we can also see that the initialization arguments MESH keeps record of changes when we reset and start the measurement object. (MESH itself doesn't interpret this list of arguments, that is up to the measurement object itself to do.)

When a measurement object is reset, MESH will internally reset the measurement reports received from the object, as well as the tide-marks associated with the object. If there are any thresholds set, they will remain set, but in such a way that triggered thresholds may once again be triggered (i.e. through the reset operation they become UT-triggered).

The measurement object is also ordered to reset itself, via the measurement type interface function `reset_measurement`, using the supplied arguments mentioned above (see also the section regarding thresholds and tide-marks).

Measurement types may be *locked*, *unlocked* and *shut down*. Below the locking capabilities and consequences are illustrated:

```
24> mesh:lock_type(diversity1).
{locked,diversity1}

25> mesh:list_types().
[{diversity1,[{extra,"A simple measurement type"},
              {interface_mod,diversity},
              {instances,0},
              {max_instances,5},
              {administrative_state,locked}]},
 {diversity2,[{extra,"Another measurement type"},
              {interface_mod,diversity},
              {instances,0},
              {max_instances,10},
              {administrative_state,unlocked}]}]

26> mesh:list_measurements(diversity1).
[]

27> mesh:create_measurement(div11,diversity1,"",0).
{error,{type_locked,diversity1}}
```

The locking of a measurement type results in the deletion of all existing measurement objects belonging to that very same type, as shown above. We also see that it is impossible to create a new measurement object, since the administrative state of the measurement type doesn't allow this operation.

We now unlock the measurement object:

```
28> mesh:unlock_type(diversity1).
{unlocked,diversity1}

29> mesh:create_measurement(div11, diversity1, "", 0).
{created,div11}

30> mesh:create_measurement(div12, diversity1, "", 0).
{created,div12}
```

Having unlocked the measurement type we may once again create measurement objects.

Finally, let us take a look at the third administrative state, *shutting down*:

```
28> mesh:shut_down_type(diversity1).
{shutting_down,diversity1}

29> mesh:list_types().
[{diversity1,[{extra,"A simple measurement type"},
```

```
              {interface_mod,diversity},
              {instances,2},
              {max_instances,5},
              {administrative_state,shutting_down}]},
 {diversity2,[{extra,"Another measurement type"},
              {interface_mod,diversity},
              {instances,0},
              {max_instances,10},
              {administrative_state,unlocked}]}]

30> mesh:create_measurement(div13, diversity1, "", 0).
{error,{type_shutting_down,diversity1}}

31> mesh:delete_measurement(div12).
{deleted,div12}

32> mesh:create_measurement(div12, diversity1, "", 0).
{error,{type_shutting_down,diversity1}}

33> mesh:delete_measurement(div11).
{deleted,div11}

34> mesh:list_types().
[{diversity1,[{extra,"A simple measurement type"},
              {interface_mod,diversity},
              {instances,0},
              {max_instances,5},
              {administrative_state,locked}]},
 {diversity2,[{extra,"Another measurement type"},
              {interface_mod,diversity},
              {instances,0},
              {max_instances,10},
              {administrative_state,unlocked}]}]
```

Note that all existing measurement objects continue to exist when shut down has been ordered, but that no new measurement objects may be created. Also, old measurement objects may not be re-created, once they have been deleted.
When the last measurement object, belonging to the type in question, has been deleted, the measurement type automatically enters state *locked*.

Deleting Measurement Objects

A measurement object is regarded as “existing” (ie. the Measurement Handler is aware of it), until it is explicitly deleted. Should the process/application implementing the measurement object crash, MESH regards the object as being *disabled* - but it still exists.
Below is an example of this, assume there is a previously created a measurement object *div11*, that unfortunately has crashed (i.e., the process `implementing` *div11* has crashed).

```
35> mesh:list_measurements(diversity1).
[{div11,[{extra,[]},
         {resources,0},
```

```
        {initial_arguments,[]},
        {operability_state,disabled},
        {administrative_state,started}]}]

36> mesh:create_measurement(div11, diversity1, "", 0).
{error,{already_created,div11}}

37> mesh:delete_measurement(div11, [4,5]).
{deleted,div11}

38> mesh:list_measurements(diversity1).
[]
```

When deleting a measurement object, MESH will call the `delete_measurement` function in the interface module. The stop arguments specified in the call to MESH (in our example, the list `[4,5]`) will be forwarded to this function. This enables a controlled and graceful termination of the measurement object (eg.leaving no open files).

Unregistering Measurement Types

Measurement types are unregistered in the following way:

```
39> mesh:unregister_type(diversity1, [4,5]).
{unregistered,diversity1}
```

Any existing measurement objects will be deleted, following the scheme described in the section above, “Deleting Measurement Objects”. In this case, to enable a controlled termination of measurement objects, stop arguments must be specified, for example `[4,5]`. These arguments will only be forwarded to the `delete_measurement` function in the measurement type interface.

Once all measurement objects are deleted, the `terminate` function in the measurement type interface is called, enabling a graceful and complete clean-up, should that be desired. This function shall terminate the MRP (if there is any), but may also be used to free common resources used by the measurement type and objects.
After this process is complete the user is free to re-use the measurement type name/identifier to associate it with another interface module.

Revival of Measurement Objects

If a measurement object becomes disabled, e.g., due to a crash in the process/application implementing it, the user may request the object revival, using the function `revive_measurement`. MESH will then order the measurement type interface to once again create the measurement function. The`create_measurement` function in the measurement type interface will be called. The arguments will be implemented according to the information MESH previously stored about the measurement object in question. The previously stored information will include the last known administration state, the last known start argument list, the resources used, the thresholds set, and so on. Provided the creation doesn’t fail, the measurement object revived will be as close as possible to a copy of the old measurement object, prior to it disabling.

Note:
The revival procedure is finished with an explicit order, issued by MESH itself, to reset the measurement object. This is to ensure identical states in the Measurement Handler and the measurement object. Even thresholds and tide-marks are reset, see description of the reset procedure.

Should revival, using the settings stored in MESH, not be wanted, the user may choose to first delete the measurement object, and then create a completely new object, using the old measurement object identifier.

Measurement Object Operation and Measurement Value Reporting

Generally, a measurement object collects a number of samples, from one or more resources and samples which later one shall be filtered using an algorithm (presumably one found in the `mesh_lib` support library). It is the sole responsibility of the user to decide *when* each sample shall be collected.

The start arguments make it possible to specify when creating, starting and resetting of measurement objects may be used to tell the measurement object when to start sampling, and the sampling interval.

In some situations it may also be desirable to dynamically tell the measurement object which filtering algorithm shall be used. However, whatever the start arguments are used for, the onus is on the user to decide how the list of arguments shall be interpreted by the measurement object.

When the measurement object has obtained a filtered measurement value, a value is reported to the Measurement Handler, using the `measurement_report` function. When the Measurement Handler receives such a measurement value report, it compares the received value with the thresholds set, sending an alarm to EVA if any threshold is triggered. It also updates the tide-marks associated with the measurement object, stores the measurement value internally, and finally sends a *meshMeasurementReport* event to EVA. This event containing the measurement value is received among other measurement specific information. (All measurement report events are recorded in a log of their own. The user/manager may later on check this log to find out the history of a certain measurement. The user may also, using the `get_measurement_report` function, ask MESH for the last measurement value received from a certain measurement).

A measurement object may also choose to report to MESH only in certain circumstances, e.g., if the user wants to specify a measurement object that just supervises a critical threshold value; this case is described in more detail below, in the section “Threshold and Tide-mark Support”.

The log used to log meshMeasurementReport events (and also the other logs used by MESH) is described below, in the section “Alarms, Events and Logs”.

Threshold and Tide-mark Support

Thresholds

The Measurement Handler supports threshold setting and control. Thresholds are a mechanism used for generation notifications, caused by changes in measured values. To avoid repeated triggering of notifications, there should be small oscillations around a threshold value, the Measurement Handler supports a so-called hysteresis mechanism.

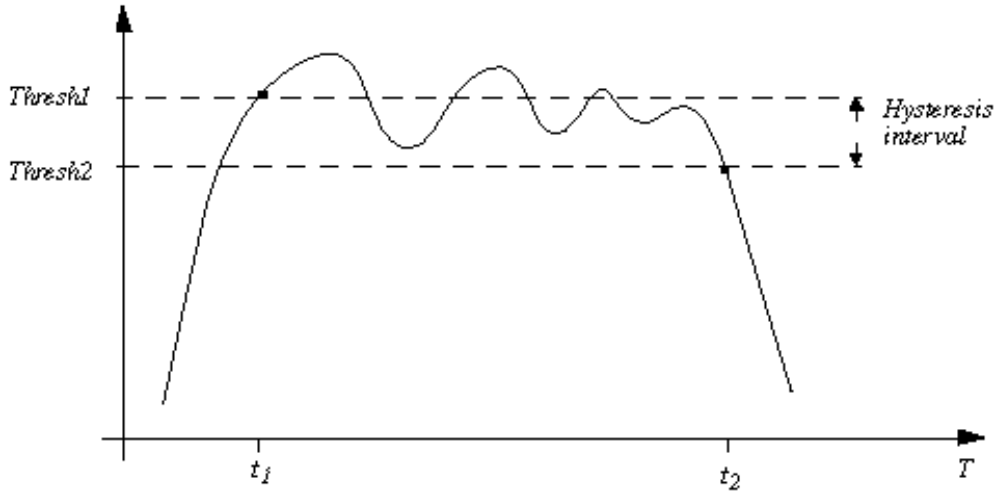

Figure 1.2: An upper threshold.

From the Measurement Handler point of view, there are two kinds of thresholds: *upper thresholds* and *lower thresholds*, the difference being whether the notification shall be triggered when the measurement value exceeds the threshold value or when it falls below the threshold value.

Regardless of type, each threshold consists of two values, in this document they are denoted `Thresh1` and `Thresh2`. On a time-scale, `Thresh1` shall be the value first encountered, as seen in the figure above.

For each measurement object, an infinite number of thresholds may be set. To distinguish between different thresholds within a given measurement object, each threshold has to have a unique identifier, assigned by the user. (An identifier may be re-used for another measurement object, and it is only within the measurement object the identifier has to be unique!)

When setting a threshold, the user may decide whether it shall be *disabled* or *enabled*. Only enabled thresholds can generate threshold notifications.

When the Measurement Handler receives a measurement value report, it compares the measurement value to each set threshold value. If the received value exceeds (or falls below, in the lower threshold case) the `Thresh1` value, an alarm is sent to EVA.
Should the received value fall below (or exceed, in the lower threshold case) the `Thresh2` value, and this threshold previously has generated an alarm, this alarm in EVA will be cleared.

When reseting a measurement object, all threshold alarms (concerning this measurement object) sent to EVA will be cleared; the next received measurement value will then be compared to the thresholds set, according to the above description.

The user may also choose to personally implement the threshold handling. For that purpose, all threshold handling functions (with the exception of the `list_thresholds` function) will cause MESH to call a corresponding function in the measurement type interface (see the `mesh` Reference Manual for complete details) and forward the same information it received. The user may then choose to silently compare measurement values to the thresholds set internally in the measurement object, and not take any action until a threshold has been triggered. The measurement object may then either send a notification directly to the manager, or just send a measurement report to MESH. The same threshold will then be triggered in MESH, resulting in an alarm being sent to EVA.

Tide-marks

A tide-mark is a mechanism that records the maximum or minimum value reached during a measurement period. A tide-mark is read-only, however it may be reset, as we will see below.

There are two kinds of tide-marks:

- the maximum tide-mark, which changes (increases) only when the associated measurement value increases beyond the current tide-mark value. The measurement value then becomes the new tide-mark value.
- the minimum tide-marks, which changes (decreases) only when the associated measurement value falls below the current tide-mark value. The measurement value then becomes the new tide-mark value.

The Measurement Handler supports, for each measurement object, both a maximum and a minimum tide-mark.

There are three components in each tide-mark:

- the `current value`. The original value is the atom `undefined`.
- the `previous current value`, i.e., the current value immediately prior to the last reset. The original value is the atom `undefined`.
- the `reset time`, i.e., the time of the last reset. The original value is the atom `undefined`.

If reset of a measurement object is ordered, both the maximum and minimum tide-mark associated with the measurement object will be reset. This is done in the following way:

- the `previous current value` is assigned the `current value`.
- the `current value` is assigned the atom `undefined`.
- the `reset time` is set to the current time and date (using the Universal Coordinated Time, UTC; sometimes called GMT).

The tide-marks may be read using the `report_tidemarks` function.

The Watchdog

The Measurement Handler supports a “watchdog”, that supervises the total number of measurement types registered, and the total number of created measurement objects. If any of these numbers exceed the maximum allowed number, a *meshTooManyTypes* or *meshTooManyMeasurements* alarm is sent to EVA (see also below).

Initially an infinite number of measurement types and measurement objects are allowed, but the user may alter this, using the `set_watchdog` function, see the `mesh` Reference Manual for complete details.

Alarms, Events and Logs

Alarms

The following alarms may be sent to EVA:

- *meshThresholdTriggered*. This alarm, belongs to the class `qos` and having severity `indeterminate`, is sent when a threshold has been triggered. The fields of interest in the alarm are as follows:

```
Field        Value
-----        -----
sender       mesh_server
cause        {upper_threshold_triggered, {value, number()}}  |
             {lower_threshold_triggered, {value, number()}}
extra        {{meas, MeasId}, {id, ThreshId}}
```

  Once the threshold gets untriggered the alarm is cleared.

- *meshTooManyTypes* , This alarm, belongs to the class `processing` and having severity `warning`, is sent when the total number of measurement types exceeds the allowed number. The fields of interest in the alarm are as follows:

```
Field        Value
-----        -----
sender       mesh_server
cause        {{allowed,number()},{currently,number()}}
extra        ""
```

  Once the current number of registered types falls below the allowed number (or the allowed number exceeds the current number) the alarm is cleared.

- *meshTooManyMeasurements*. This alarm, belongs to the class `processing` and having severity `warning`, is sent when the total number of measurement objects exceeds the allowed number. The fields of interest in the alarm are as follows:

```
Field        Value
-----        -----
sender       mesh_server
cause        {{allowed,number()}, {currently,number()}}
extra        ""
```

  Once the current number of measurement objects falls below the allowed number (or the allowed number exceeds the current number) the alarm is cleared.

- *meshTypeCapacityExceeded*. This alarm, belongs to the class `processing` and having severity `warning`, is sent when the number of measurement objects exceeds the number allowed for the measurement type they belong to (due to a capacity decrease for the measurement type in question). The fields of interest in the alarm is as follows:

```
Field        Value
-----        -----
sender       mesh_server
cause        {{type,TypeId}, {allowed,number()}, {currently,number()}}
extra        "Capacity decreased"
```

Once the capacity is sufficiently increased, or a sufficient number of measurement objects have been deleted, the alarm will cleared.

Events

The following events may be sent to EVA:

- *meshTypeFailure*. This event is sent when an MRP associated with a measurement type (a common MRP) has terminated. The `extra` field in the event will contain the following information: {`MeasId, FailureReason, FailureTime`}.
- *meshMeasurementTerminated*. This event is sent when a measurement object has terminated. The `extra` field in the event will contain the following information: {`MeasId, TypeId, TerminationReason, TerminationTime`}.
- *meshNodeUp*. This event is sent when a crashed node, known to have had at least one MRP, is connected again. The `extra` field in the event will contain the following information: {`NodeName, ConnectionTime`}.
- *meshNodeDown*. This event is sent when a node, having at least one MRP, has crashed. The `extra` field in the event will contain the following information: {`NodeName, CrashTime`}.
- *meshTypeUnconnected*. This event is sent when a node connection with (at least) one MRP, has been lost, but the status of the MRP is yet unknown. The `extra` field in the event will contain the following information: {`TypeId, nodedown, Time`}.
- *meshMeasurementUnconnected*. This event is sent when the node connection, with (at least) one measurement object, has been lost but the status of the measurement object is yet unknown. The `extra` field in the event will contain the following information: {`MeasId, TypeId, nodedown, Time`}.
- *meshTypeConnected*. This event is sent when the connection to a node has been restored, and the MRP is found to still exist. The `extra` field in the event will contain the following information: {`TypeId, nodeup, ConnectionTime`}.
- *meshMeasurementConnected*. This event is sent when the connection to a node has been restored, and the measurement object is found to still exist. The `extra` field in the event will contain the following information: {`MeasId, TypeId, nodeup, ConnectionTime`}.
- *meshMeasurementReport*. This event is sent when the Measurement Handler has received a measurement report from a measurement object. The `extra` field in the event will contain the following information: {{`name,MeasId`}, {`value,number()`}, {`time,Time`}, {`info,MeasInfo`}}.

Logs

The following logs are used by the Measurement Handler:

- *mesh_alarms*. In this log all alarms sent, by MESH to EVA, are logged.
- *mesh_events*. In this log all events sent, by MESH to EVA, are logged. The only exception is the `meshMeasurementReport` events, which are recorded in their own log, see below.
- *mesh_measurements*. In this log all `meshMeasurementReport` events are logged.

Should any of the logs mentioned above not be available, the default log (that EVA is started with) will be used instead.

By default, each log has a filter that ensures that only the required elements are logged. MESH supports the switching of logs, should the user want to use their own filter function (for extended functionality). The functions available are `set_alarm_filter`, `set_event_filter` and `set_measurement_filter`, please see the `mesh` Reference Manual for further details. We recommend cross referencing with the `eva` Reference Manual.
(Should the user decide later to reset the filters to the original settings, the functions `reset_alarm_filter`, `reset_event_filter` and `reset_measurement_filter` are available.)

1.3 MESH SNMP interface

This chapter describes a MESH adaptation for SNMP.

There is one MIB defined: `OTP-MESH-MIB`. This MIB can be found in the `mibs` directory in the MESH distribution. It is more closely described in the following sections.

MESH SNMP adaptation

The MESH SNMP adaptation consists of functionality for translating the MESH events and alarms to SNMP traps, an SNMP MIB used to interface the MESH application remotely, and an API to be used when modifying the adaptation in a desired way.

OTP-MESH-MIB

This MIB implements managed objects for the basic MESH service in OTP. It consists of five tables, five variables, and a number of trap definitions.
The OTP-MESH-MIB is written according to SNMPv2.

The typeTable The `typeTable` has one entry for every measurement type known to the system.

The `typeTable` consists of measurement type attributes that are both readable and writable. Through this table the manager may register measurement types, unregister them, and change their administrative state.

The table has the following attributes:

- `typeName`
- `typeInfo`
- `typeCallbackMod`
- `typeAdminState`
- `typeMeasArgs`
- `typeMaxInst`
- `typeStatus`

Each measurement type has a unique index, `typeName`, which remains constant as long as the measurement type entry isn't explicitly destroyed. This index is a `DisplayString`, which will be translated to an atom inside the MESH SNMP adaptation. That is, all measurement type names have to be atoms, should the MESH SNMP adaptation be used!

The `typeInfo` attribute may contain any information describing the measurement type.
The default value of this attribute is the empty list ("").

The `typeCallbackMod` attribute defines the interface type module to use when communicating with the measurement type in question.

The `typeAdminState` defines the current administrative state of the measurement type, i.e., whether it shall be `locked`, `unlocked` or `shutting_down`. These values are enumerated: 1 == `unlocked`, 2 == `shutting_down` and 3 == `locked`.

The default value is `unlocked`.

The `typeMeasArgs` specifies the arguments that, when the type is locked, shall be forwarded to the (active) measurement objects belonging to the specified measurement type. It shall be noted that these arguments are ONLY used when the type is locked, never else!
The arguments are given as an `DisplayString`, which is translated by the MESH SNMP adaptation to an Erlang term. That is, if one wants the list `[1,2,3,4]` to be forwarded to the measurement objects, this attribute shall be set to `"[1,2,3,4]"`.
The default value is the empty list ([]).

The `typeMaxInst` specifies the number of measurement objects that are allowed within this type.
The default value is 50.

The `typeStatus` attribute is used to create new entrys (i.e., to register measurement types), and to change the state of existing entries.
It shall be noted that only active entries corresponds to registered measurement types!

Examples We assume the system is up and running, and that the MESH SNMP has been started according to the description found in the earlier chapter “Services”. We also assume we have started an SNMP agent and a manager, and that we have loaded the OTP-MESH-MIB in both of them, as well as the MIBs EVA requires.
Now assume we want to register a measurement type called `diversity1`, which uses the interface module `diversity`. The original administrative state shall be `unlocked`, and we allow 5 measurement objects to be started within this type:

```
1> snmp_mgr:s([{[typeInfo | "diversity1"],"A simple measurement type"},
               {[typeCallbackMod | "diversity1"],"diversity"},
               {[typeAdminState | "diversity1"], 1},
               {[typeMaxInst | "diversity1"],5},
               {[typeStatus | "diversity1"], 4}]).
ok
* Got PDU:
Response,         Request Id:209647922
  [typeInfo,100,105,118,101,114,115,105,116,121,49] = "A simple measurement type"
  [typeCallbackMod,100,105,118,101,114,115,105,116,121,49] = "diversity"
  [typeAdminState,100,105,118,101,114,115,105,116,121,49] = 1
  [typeMaxInst,100,105,118,101,114,115,105,116,121,49] = 5
  [typeStatus,100,105,118,101,114,115,105,116,121,49] = 4
```

And if we now look at this new entry:

```
2> snmp_mgr:gn([[typeName,""]]).
ok
* Got PDU:
Response,         Request Id:223586022
  [typeInfo,100,105,118,101,114,115,105,116,121,49] = "A simple measurement type"

3> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:66538327
  [typeCallbackMod,100,105,118,101,114,115,105,116,121,49] = "diversity"
```

```
4> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:166382073
  [typeAdminState,100,105,118,101,114,115,105,116,121,49] = 1

5> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:118759047
  [typeMeasArgs,100,105,118,101,114,115,105,116,121,49] = []

6> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:215999470
  [typeMaxInst,100,105,118,101,114,115,105,116,121,49] = 5

7> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:254918516
  [typeStatus,100,105,118,101,114,115,105,116,121,49] = 1
```

We may also decide to set the administrative state to `locked`:

```
8> snmp_mgr:s([{[typeAdminState | "diversity1"], 3}]).
ok
* Got PDU:
Response,         Request Id:14458468
  [typeAdminState,100,105,118,101,114,115,105,116,121,49] = 3
```

And if we want to unregister the type and remove it from the system:

```
9> snmp_mgr:s([{[typeStatus | "diversity1"], 6}]).
ok
* Got PDU:
Response,         Request Id:186830689
  [typeStatus,100,105,118,101,114,115,105,116,121,49] = 6
```

The typeInfoTable The `typeInfoTable` has one entry for every measurement type known to the system.

The `typeInfoTable` consists of measurement type attributes that are readable only.

The table has the following attributes:

- `typeInfoName`
- `typeInfoCurrInst`

The typeInfoName is the index field; each measurement type corresponds to one entry. This index remains constant as long as the measurement type entry isn't explicitly destroyed. This index is a DisplayString, which will be translated to an atom inside the MESH SNMP adaptation.

The typeInfoCurrInst attribute contains the current number of active measurement objects belonging to this specific measurement type.

The measTable The measTable has one entry for every measurement object known to the system.

The measTable consists of measurement object attributes that are both readable and writable. Through this table the manager may create measurement objects, delete them, and change their administrative state (i.e., start and stop them).

The table has the following attributes:

- measId
- measType
- measInfo
- measResId
- measAdminState
- measArgs
- measStatus

Each measurement object has a unique index, measId, which remains constant as long as the measurement object entry isn't explicitly destroyed. This index is a DisplayString, which will be translated to an atom inside the MESH SNMP adaptation. That is, all measurement object names have to be atoms, should the MESH SNMP adaptation be used!

The measType attribute specifies the measurement type the measurement object belongs to. This attribute is also given as a DisplayString, which will be translated to an atom inside the MESH SNMP adaptation.
Note that the specified measurement type has to be registered, otherwise the create operation will fail!

The measInfo attribute may contain any information describing the measurement object.
The default value of this attribute is the empty list ("").

The measResId attribute is a DisplayString specifying the resources the measurement object is assumed to use. This string will be translated to an Erlang term inside the MESH SNMP adaptation.

The measAdminState defines the current administrative state of the measurement object, i.e., whether it shall be started or stopped. These values are enumerated: 1 == started and 2 == stopped.
The default value is started.

The measArgs specifies the arguments that, when the mesurement object is created, deleted, started, stopped, or reset, shall be forwarded to it. It shall be noted that these arguments are NOT used when the type is locked; in that case the argument list specified in the typeTable is used instead! The arguments are given as an DisplayString, which is translated by the MESH SNMP adaptation to an Erlang term. That is, if one wants the list [1,2,3,4] to be forwarded to the measurement object, this attribute shall be set to "[1,2,3,4]".
The default value is the empty list ([]).

The measStatus attribute is used to create new entrys (i.e., to create new measurement objects), and to change the state of existing entries.
It shall be noted that only active entries correspond to created measurement objects!

Examples We assume the same system state as in the previous example, and also that we have registered a measurement type “diversity1”.
Now assume we want to create a measurement object called `div11`. This object shall use a resource identified with the integer 0 (zero), the original administrative state shall be `started`, and the initial arguments shall be the list `[1,2,3]`:

```
10> snmp_mgr:s([{[measType | "div11"],"diversity1"},
                {[measInfo | "div11"],"A simple measurement object"},
                {[measResId | "div11"],"0"},
                {[measAdminState | "div11"],1},
                {[measArgs | "div11"],"[1,2,3]"},
                {[measStatus | "div11"],4}]).
ok
* Got PDU:
Response,         Request Id:12669132
  [measType,100,105,118,49,49] = "diversity1"
  [measInfo,100,105,118,49,49] = "A simple measurement object"
  [measResId,100,105,118,49,49] = "0"
  [measAdminState,100,105,118,49,49] = 1
  [measArgs,100,105,118,49,49] = "[1,2,3]"
  [measStatus,100,105,118,49,49] = 4
```

And if we now look at this new entry:

```
11> snmp_mgr:gn([[measId,""]]).
ok
* Got PDU:
Response,         Request Id:115469191
  [measType,100,105,118,49,49] = "diversity1"

12> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:139112371
  [measInfo,100,105,118,49,49] = "A simple measurement object"

13> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:164740948
  [measResId,100,105,118,49,49] = "0"

14> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:76656359
  [measAdminState,100,105,118,49,49] = 1

15> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:105654964
```

```
  [measArgs,100,105,118,49,49] = [1,2,3]

16> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:59931249
  [measStatus,100,105,118,49,49] = 1
```

We may also look at the updated measurement type information:

```
17> snmp_mgr:g([[typeInfoCurrInst | "diversity1"]]).
ok
* Got PDU:
Response,         Request Id:252519077
  [typeInfoCurrInst,100,105,118,101,114,115,105,116,121,49] = 1
```

We may stop the measurement object:

```
18> snmp_mgr:s([{[measAdminState | "div11"],2}]).
ok
* Got PDU:
Response,         Request Id:262901542
  [measAdminState,100,105,118,49,49] = 2
```

And we may start the object again, with some new arguments:

```
19> snmp_mgr:s([{[measAdminState | "div11"],1},{[measArgs | "div11"],"[4,5,6]"}]).
ok
* Got PDU:
Response,         Request Id:257409456
  [measAdminState,100,105,118,49,49] = 1
  [measArgs,100,105,118,49,49] = "[4,5,6]"
```

And finally we may also delete our measurement object, supplying the string “free_resources” as terminating arguments (remember that this string will be interpreted as an Erlang term in the MESH SNMP adaptation - in this case as an atom!):

```
20> snmp_mgr:s([{[measArgs | "div11"],"free_resources"},{[measStatus | "div11"], 6}]).
ok
* Got PDU:
Response,         Request Id:152978686
  [measArgs,100,105,118,49,49] = "free_resources"
  [measStatus,100,105,118,49,49] = 6
```

To convince ourselves that the measurement object has disappeared, we may look at the current number of instances belonging to measurement type `diversity1`:

```
21> snmp_mgr:g([[typeInfoCurrInst | "diversity1"]]).
ok
* Got PDU:
Response,         Request Id:136894792
  [typeInfoCurrInst,100,105,118,101,114,115,105,116,121,49] = 0
```

The measInfoTable The measInfoTable has one entry for every measurement object known to the system.

The typeInfoTable consists of measurement object attributes that are readable only.

The table has the following attributes:

- measInfoId
- measInfoLastVal
- measInfoLastValTime
- measInfoLastValInfo
- measInfoMaxTideCurr
- measInfoMaxTidePrev
- measInfoMinTideCurr
- measInfoMinTidePrev
- measInfoLastReset

The MeasInfoId is the index field; each measurement object corresponds to one entry. This index remains constant as long as the measurement object entry isn't explicitly destroyed. This index is a DisplayString, which will be translated to an atom inside the MESH SNMP adaptation.

The measInfoLastVal attribute contains, as a DisplayString, the last measurement value the Measurement Handler has received from the specified measurement object.

The measInfoLastValTime attribute contains, as a DisplayString, the time the last measurement value was obtained by the measurement object.

The measInfoLastValInfo attribute contains, as a DisplayString, additional information about the the last measurement value the Measurement Handler has received.

The measInfoMaxTideCurr attribute contains, as a DisplayString, the current value in the maximum tidemark associated with the specified measurement object.

The measInfoMaxTidePrev attribute contains, as a DisplayString, the previous value in the maximum tidemark associated with the specified measurement object.

The measInfoMinTideCurr attribute contains, as a DisplayString, the current value in the minimum tidemark associated with the specified measurement object.

The measInfoMinTidePrev attribute contains, as a DisplayString, the previous value in the minimum tidemark associated with the specified measurement object.

The measInfoLastReset attribute contains, using the DateAndTime format (see MIB SNMPv2-TC), the date and time of the last reset.

Note:
If the measurement object never has been reset, the list [0,0,0,0,0,0,0,0] will be returned.

Examples We assume that we have the measurement object `div11` (see previous example) running, that is, reporting measurement values, and also that this measurement object has been reset once.

```
22> snmp_mgr:gn([[measInfoId, ""]]).
ok
* Got PDU:
Response,         Request Id:256037314
  [measInfoLastVal,100,105,118,49,49] = "100"

23> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:244235445
  [measInfoLastValTime,100,105,118,49,49] = "{19,52,37}"

24> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:141582032
  [measInfoLastValInfo,100,105,118,49,49] = []

25> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:182241904
  [measInfoMaxTideCurr,100,105,118,49,49] = "100"

26> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:211213648
  [measInfoMaxTidePrev,100,105,118,49,49] = "100"

27> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:210013650
  [measInfoMinTideCurr,100,105,118,49,49] = "0"

28> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:2594486
  [measInfoMinTidePrev,100,105,118,49,49] = "0"

29> snmp_mgr:gn().
ok
* Got PDU:
Response,         Request Id:46299549
  [measInfoLastReset,100,105,118,49,49] = [19,98,7,8,17,52,33,0]
```

The threshTable The threshTable has one entry for every threshold known to the system.

The threshTable consists of threshold attributes that are both readable and writable. Through this table the manager may set upper and lower thresholds, remove them, and change their administrative state (i.e., enable and disable them).

The table has the following attributes:

- threshMeasId
- threshId
- threshType
- threshVal1
- threshVal2
- threshAdminState
- threshStatus

The threshMeasId and the threshId fields are the indices; this is due to the fact that a threshold is identified uniquely by the threshold identifier together with the measurement object it belongs to. Each threshold in the system corresponds to one entry. The two indices remain constant as long as the threshold entry isn't explicitly destroyed.
The threshMeasId index field is a DisplayString, which will be translated to an atom inside the MESH SNMP adaptation. The threshId index field is an integer.

Note:
NOTE: All threshold identifiers have to be integers, should the MESH SNMP adaptation be used!

The threshType attribute decides whether the threshold is an upper threshold or a lower one. These two types are enumerated: 1 == upper and 2 == lower.
The default value is upper.

The threshVal1 attributes sets the threshold value that triggers the threshold. The value is given as a DisplayString, which is converted to an Erlang term in the MESH SNMP adaptation.
The default value is "0" (zero).

The threshVal2 attributes sets the threshold value that un-triggers the threshold. The value is given as a DisplayString, which is converted to an Erlang term in the MESH SNMP adaptation.
The default value is the one that threshVal1 was set to.

The threshAdminState tells whether the threshold shall be enabled or disabled. These states are enumerated: 1 == enabled and 2 == disabled. The default value is enabled.

The threshStatus attribute is used to create new entrys (i.e., to set new thresholds), and to change the state of existing entries.
It shall be noted that only active entries correspond to set thresholds!

Examples We assume that we have the same measurement object `div11` as in the previous example. We want to set an upper threshold, having the integer 1 as identifier, and with original administrative state `enabled`. After the creation we disable it, enable it again, and finally deletes it:

```
30> snmp_mgr:s([{[threshType | "div11" ++ [1]], 1},
                {[threshVal1 | "div11" ++ [1]], "90"},
                {[threshVal2 | "div11" ++ [1]], "85"},
                {[threshAdminState | "div11" ++ [1]], 1},
                {[threshStatus | "div11" ++ [1]], 4}]).
ok
* Got PDU:
Response,         Request Id:124268928
  [threshType,100,105,118,49,49,1] = 1
  [threshVal1,100,105,118,49,49,1] = "90"
  [threshVal2,100,105,118,49,49,1] = "85"
  [threshAdminState,100,105,118,49,49,1] = 1
  [threshStatus,100,105,118,49,49,1] = 4

31> snmp_mgr:s([{[threshAdminState | "div11" ++ [1]], 2}]).
ok
* Got PDU:
Response,         Request Id:165662114
  [threshAdminState,100,105,118,49,49,1] = 2

32> snmp_mgr:s([{[threshAdminState | "div11" ++ [1]], 1}]).
ok
* Got PDU:
Response,         Request Id:225312294
  [threshAdminState,100,105,118,49,49,1] = 1

33> snmp_mgr:s([{[threshStatus | "div11" ++ [1]], 6}]).
ok
* Got PDU:
Response,         Request Id:153618962
  [threshStatus,100,105,118,49,49,1] = 6
```

The watchdogNofTypes variable This variable is used to set the allowed (total) number of measurement types that are allowed in the system.

The watchdogNofMeas variable This variable is used to set the allowed (total) number of measurement objects that are allowed in the system.

The currentNofTypes variable This variable is used to read the current (total) number of measurement types in the system.

The currentNofMeas variable This variable is used to read the current (total) number of measurement objects in the system.

The currentNofMeas variable This variable is used to reset measurement objects. The measurement object is entered as a `DisplayString`, which is interpreted as an atom inside the MESH SNMP adaptation. Reset arguments are taken from the corresponding `measArgs` field in the `measTable`. That is, the `measArgs` field has to be set to the desired value BEFORE this variable is set!

Note:
The reset operation is allowed in all measTable row/measurement object states, as long as the row exists, but the operation is only meaningful (i.e., will only have any real effect) when the row state is 'active'.

Note:
This variable is intended to be write-only; however, this is not allowed in SNMP, meaning the variable has to be read-write. If the user tries to read the current value, it shall be noted that it is only the last reset request made using SNMP that is shown! That is, measurement objects/applications/supervisors may, in the managed system, have ordered reset themselves - these reset orders are NOT visible through this variable!

1.4 Measurement Handler Release Notes

This document describes the changes made to the MESH application.

MESH - Measurement Handler v1.1.0

Improvements and New Features

- An SNMP adaptation has been added.

Fixed Bugs and Malfunctions

-

Incompatibilities With Measurement Handler v1.0.1

- If the SNMP adaptation shall be used, measurement types and measurement objects shall be identified using atoms, and thresholds using an integer.

Known Bugs and Problems

-

MESH - Measurement Handler v1.0.1

Improvements and New Features

- A config_change function has been added to the application callback module.

Fixed Bugs and Malfunctions

-

Incompatibilities With Measurement Handler v1.0.0

-

Known Bugs and Problems

-

MESH - Measurement Handler v1.0.0

New application.

Measurement Handler (MESH) Reference Manual

Short Summaries

mesh

No functions are exported

mesh

The following functions are exported:

- `lock_type(TypeId) -> Result`
  [page 52] Locks the specified measurement type, ie. sets its administrative state to `locked`.
- `lock_type(TypeId, StopArgs) -> Result`
  [page 52] Locks the specified measurement type, ie. sets its administrative state to `locked`.
- `unlock_type(TypeId) -> Result`
  [page 52] Unlocks the specified measurement type, ie. sets its administrative state to `unlocked`.
- `shut_down_type(TypeId) -> Result`
  [page 53] Shuts down the specified measurement type, ie. sets its administrative state to `shutting down`.
- `create_measurement(MeasId, TypeId, Extra, ResId) -> Result`
  [page 53] Creates a new measurement object, belonging to the specified measurement type.
- `create_measurement(MeasId, TypeId, Extra, ResId, AdminState) -> Result`
  [page 53] Creates a new measurement object, belonging to the specified measurement type.
- `create_measurement(MeasId, TypeId, Extra, ResId, AdminState, StartArgs) -> Result`
  [page 53] Creates a new measurement object, belonging to the specified measurement type.
- `delete_measurement(MeasId) -> Result`
  [page 54] Deletes a previously created measurement object.
- `delete_measurement(MeasId, StopArgs) -> Result`
  [page 54] Deletes a previously created measurement object.
- `measurement_terminated(MeasId, Reason) -> ok`
  [page 54] Tells the MESH application about terminated measurement objects.
- `list_measurements(TypeId) -> Result`
  [page 54] Lists all created measurement objects belonging to the specified measurement type.
- `revive_measurement(MeasId) -> Result`
  [page 55] Tries to recreate a previously created but disabled, ie. terminated, measurement object. The last known settings will be used.
- `start_measurement(MeasId) -> Result`
  [page 55] Orders the specified measurement object to start running (ie. sets its administrative state to `unlocked`).
- `start_measurement(MeasId, StartArgs) -> Result`
  [page 55] Orders the specified measurement object to start running (ie. sets its administrative state to `unlocked`).
- `stop_measurement(MeasId) -> Result`
  [page 56] Orders the specified measurement object to stop running. (ie. sets its administrative state to `locked`).
- `reset_measurement(MeasId) -> Result`
  [page 56] Orders the specified measurement object to reset, and also resets internally everything concerning that very same measurement object.

- `reset_measurement(MeasId, StartArgs) -> Result`
  [page 56] Orders the specified measurement object to reset, and also resets internally everything concerning that very same measurement object.
- `measurement_report(MeasId, Value, TimeStamp) -> ok`
  [page 56] A measurement value report, from the specified measurement object, is sent to the Measurement Handler.
- `measurement_report(MeasId, Value, TimeStamp, MeasInfo) -> ok`
  [page 56] A measurement value report, from the specified measurement object, is sent to the Measurement Handler.
- `get_measurement_report(MeasId) -> Result`
  [page 57] Retrieves from the Measurement Handler, the last received measurement value from the specified measurement object.
- `set_upper_threshold(MeasId, ThreshId, Value) -> Result`
  [page 57] Sets an upper threshold associated with the specified measurement object.
- `set_upper_threshold(MeasId, ThreshId, Value, Status) -> Result`
  [page 57] Sets an upper threshold associated with the specified measurement object.
- `set_lower_threshold(MeasId, ThreshId, Value) -> Result`
  [page 58] Sets a lower threshold associated with the specified measurement object.
- `set_lower_threshold(MeasId, ThreshId, Value, Status) -> Result`
  [page 58] Sets a lower threshold associated with the specified measurement object.
- `remove_threshold(MeasId, ThreshId) -> Result`
  [page 59] Removes the threshold specified by the threshold identifier, from a measurement object.
- `remove_thresholds(MeasId) -> Result`
  [page 59] Removes all thresholds from specified measurement objects.
- `list_thresholds(MeasId) -> Result`
  [page 59] Lists all thresholds set for the specified measurement object.
- `enable_threshold(MeasId, ThreshId) -> Result`
  [page 60] Enables a previously set threshold.
- `disable_threshold(MeasId, ThreshId) -> Result`
  [page 60] Disables a previously set threshold.
- `report_tidemarks(MeasId) -> Result`
  [page 60] Returns the tidemarks associated with the specified measurement object.
- `reset_tidemarks(MeasId) -> Result`
  [page 61] Resets the tidemarks associated with a specified measurement object.
- `watchdog_setup(NofTypes, NofMeas) -> Result`
  [page 61] Settings for the watchdog supervising the total number of registered measurement types and created measurement objects.
- `set_alarm_filter(Func) -> Result`
  [page 62] Sets a new filter to be used when logging alarms in the `mesh_alarms` log.
- `set_event_filter(Func) -> Result`
  [page 62] Sets a new filter to be used when logging alarms in the `mesh_events` log.
- `set_measurement_filter(Func) -> Result`
  [page 62] Sets a new filter to be used when logging alarms in the `mesh_measurements` log.

- `reset_alarm_filter() -> Result`
  [page 63] Resets the filter to be used, when logging alarms in the `mesh_alarms` log, to the filter provided by the MESH application.
- `reset_event_filter() -> Result`
  [page 63] Resets the filter to be used, when logging alarms in the `mesh_events` log, to the filter provided by the MESH application.
- `reset_measurement_filter() -> Result`
  [page 63] Resets the filter to be used, when logging alarms in the `mesh_measurements` log, to the filter provided by the MESH application.
- `InterfaceMod:init(TypeId) -> Result`
  [page 64] Initialization function when registering a measurement type.
- `InterfaceMod:terminate(MRP) -> Result`
  [page 64] Termination function, used for clean up purposes when a measurement type is unregistered.
- `InterfaceMod:create_measurement(MRP1, TypeId, MeasId, ResId, StartArgs) -> Result`
  [page 64] Orders the measurement type interface to create a new measurement object.
- `InterfaceMod:start_measurement(MRP, MeasId, StartArgs) -> Result`
  [page 64] Orders the measurement type interface to start the specified measurement object.
- `InterfaceMod:stop_measurement(MRP, MeasId) -> Result`
  [page 65] Orders the measurement type interface to stop the specified measurement object.
- `InterfaceMod:reset_measurement(MRP, MeasId, StartArgs) -> Result`
  [page 65] Orders the measurement type interface to reset the specified measurement object.
- `InterfaceMod:delete_measurement(MRP, MeasId, StopArgs) -> Result`
  [page 65] Orders the measurement type interface to delete the specified measurement object.
- `InterfaceMod:set_upper_threshold(MRP, MeasId, ThreshId, Value, Status) -> Result`
  [page 65] Orders the measurement type interface to set the specified upper threshold in the specified measurement object.
- `InterfaceMod:set_lower_threshold(MRP, MeasId, ThreshId, Value, Status) -> Result`
  [page 66] Orders the measurement type interface to set the specified lower threshold in the specified measurement object.
- `InterfaceMod:remove_threshold(MRP, MeasId, ThreshId) -> Result`
  [page 66] Orders the measurement type interface to remove the specified threshold in a measurement object.
- `InterfaceMod:enable_threshold(MRP, MeasId, ThreshId) -> Result`
  [page 66] Orders the measurement type interface to enable the specified threshold in the specified measurement object.
- `InterfaceMod:disable_threshold(MRP, MeasId, ThreshId) -> Result`
  [page 67] Orders the measurement type interface to disable the specified threshold in the specified measurement object.

mesh_lib

The following functions are exported:

- `sum(SampleList) -> Result`
  [page 68] Sums a list of samples.
- `sum_and_squaresum(SampleList) -> Result`
  [page 68] Sums a list of samples, and also sums the square of all samples.
- `sample_mean(SampleList) -> Result`
  [page 68] Equates the mean of a list of samples.
- `sample_variance(SampleList) -> Result`
  [page 68] Equates the variance of a list of samples.
- `sample_mean_and_variance(SampleList) -> Result`
  [page 69] Equates the mean and the variance of a list of samples.
- `mean_variance(MeanList) -> Result`
  [page 69] Equates the variance of a list of estimated means.
- `ewma_mean(Xnew, Wn, GP, MTP) -> Result`
  [page 69] Computes the mean of a number of samples using the Exponentially Weighted Moving Average technique.
- `ewma_variance(Xnew, Wnew, Sn, GP, SMTP) -> Result`
  [page 70] Computes the variance of a number of samples using the Exponentially Weighted Moving Average technique.
- `uwma_mean(Xnew, SX, Xold, N) -> Result`
  [page 70] Calculaes the mean of a number of samples using the Uniformly Weighted Moving Average technique.
- `uwma_variance(Xnew, SX, SqSX, Xold, N) -> Result`
  [page 70] Computes the variance of a number of samples using the Uniformly Weighted Moving Average technique.

mesh_snmp

The following functions are exported:

- `register_alarms(Community) -> void()`
  [page 72] Registers the alarms used by the MESH application.
- `register_events(Community) -> void()`
  [page 72] Registers the events used by the MESH application.

mesh (Application)

The Measurement Handler appliction (MESH) is an Operation and Maintenance management application that firstly, provides support to applications and managers for creation and control of measurement types and objects, and secondly, allows control of the associated logs.

MESH is designed to be a distributed global application, which means that although it may only be running at one node at a time, with the other nodes as standby nodes, clients may access the MESH functionality from any node.

The MESH API can be used to make management protocol specific interfaces to MESH, for example SNMP, CORBA, or HTTP interfaces. Currently, an SNMP interface is included in the application; others may follow.

MESH uses the Mnesia DBMS to store data. This means that Mnesia must be running on all nodes where MESH may run, and that the tables MESH uses must be created and configured correctly. MESH provides a function that should be called to create tables, and to define replicas for Mnesia. This function is called `create_tables`, and takes one parameter, a list of nodes. The Mnesia tables will be replicated on these nodes; some on disk, and some in RAM. It is important that these nodes are the same as those where MESH will run.

MESH also uses EVA functionality to send and log generated events and alarms. The logs can be examined by a manager at a later time, however, the configuration and start of EVA is left to the client application programmer (please see the EVA User's Guide and manual pages).

Commonly used statistical functions may be found in the module `mesh_lib`.

CONFIGURATION

The following configuration parameters are defined for the MESH application (for more information about configuration parameters see application(3)):

`snmp_adapted = true | false` <optional> Specifies whether or not the MESH SNMP adaptation shall be started. The default is `false`, meaning that the adaptation isn't started.

SEE ALSO

mesh(3), mesh_lib(3), eva(3), eva_sup(3)

mesh (Module)

This module contains the client API to the Measurement Handler application MESH. MESH is a distributed global application, which means that clients can access the MESH functionality from any connected node. There is a globally registered server called `mesh_server` to which all requests are sent. Currently all functions (except the `measurement_report` and `measurement_terminated` functions) are synchronous. This means that the function call fails if no acknowledgment is returned by the server. Should this happen, the client application has to decide what actions to take. (It may, for example, wait a few seconds for another node to take over the MESH application, and then try again.)

MESH stores data about measurement types and measurement objects in the Mnesia tables `mesh_type` and `mesh_meas`, respectively. They are replicated to disk and RAM on each node that may run the MESH application.

MESH uses an EVA functionality to send and to log generated events and alarms. The logs used can be examined by a manager at a later time.
However, the configuration and start of EVA is left to the client application programmer, see section `Requirements` below. MESH doesn't supervise EVA, which essentially means that the client application programmer must be aware of EVA. For performance reasons, it is desired that EVA runs on the same node as the Measurement Handler to reduce internal traffic.

Commonly used statistical functions may be found in the module `mesh_lib`.

Starting and Stopping MESH

The MESH application is controlled through the `application` module API; MESH is started by the function call `application:start(mesh)`,
and stopped by the function call
`application:stop(mesh)`.

Requirements

The MESH application requires the following applications to be started:

- SASL
- Mnesia
- EVA

When starting Mnesia, a schema must be created on each of the nodes where the MESH application will be running; see for example function mnesia:create_schema/1.

Before EVA is started, the Mnesia tables needed by the basic EVA service and the EVA log service have to be created on each node where the MESH application will be running; see example function eva_sup:create_tables_log/1.
Also, EVA should be started with a default log, which first must be created, see for example functions disk_log:open/1 and eva:start_link_log/1.

Before MESH is started, the Mnesia tables needed by the MESH application have to be created on each of the nodes where the MESH application is supposed to be will be running; see example function mesh:create_tables/1.

Call back Module Usage

While the Measurement Handler application takes care of common tasks, user dependent, implementation specific issues are handled through the measurement type interface modules. These modules are (through the measurement type registration) associated with a specific measurement type. When MESH is ordered to perform certain tasks, concerning a certain measurement type, one or more of the functions in the associated interface module may be called.

The functions that have to be present in each interface module are (please also see the section Interface Module Functions):

- init/1
- terminate/1
- create_measurement/3
- delete_measurement/3
- start_measurement/3
- stop_measurement/2
- reset_measurement/3

The following functions are optional for each interface module:

- set_upper_threshold/5
- set_lower_threshold/5
- remove_threshold/3
- enable_threshold/3
- disable_threshold/3

The purpose of these latter functions are to provide the user with the possibility to implement the threshold supervising functionality. For example, a measurement object may just supervise a measurement value, reporting nothing until a threshold has been triggered. The measurement object in question may then report directly to the manager or EVA. Alternatively, the measurement object can send a normal measurement report to MESH, containing the measurement value that triggered the threshold. In the latter case the threshold set in MESH will be triggered, and an alarm will be sent to EVA.

The figure below describes how function calls to MESH result (or may result) in calls to the corresponding measurement type interface.

Sometimes a function may be called more than once, for example, the `delete_measurement` in the interface module is called for each measurement object. This is not indicated in the figure. Similarly, sometimes a function may `not` be called, depending on the context; this is also not indicated.

```
MESH                                Interface Module
----                                ----------------

mesh:register_type          ---->   IM:init/1

mesh:unregister_type        ---->   IM:delete_measurement
                                    IM:terminate

mesh:lock_type              ---->   IM:delete_measurement

mesh:create_measurement     ---->   IM:create_measurement

mesh:delete_measurement     ---->   IM:delete_measurement

mesh:revive_measurement     ---->   IM:create_measurement

mesh_start_measurement      ---->   IM:start_measurement

mesh:stop_measurement       ---->   IM:stop_measurement

mesh:reset_measurement      ---->   IM:reset_measurement

mesh:set_upper_threshold    ---->   IM:set_upper_threshold

mesh:set_lower_threshold    ---->   IM:set_lower_threshold

mesh:remove_threshold       ---->   IM:remove_threshold

mesh:enable_threshold       ---->   IM:enable_threshold

mesh:disable_threshold      ---->   IM:disable_threshold
```

Measurement Responsible Processes

A Measurement Responsible Process (MRP) is one or more processes enabling communication between the measurement type interface and the actual processes implementing, eg. measurement objects. Since the measurement type interface is stateless, it has to be supplied with the identifiers of these processes, which is done by the Measurement Handler, whenever it calls a measurement type interface function. The measurement type interface must ensure the correct process is contacted.

Basically, there are two MRP design possibilities:

- each measurement object is its own MRP.
- one process, working as a server, keeps track of one or more measurement objects, forwarding the messages received to the correct measurement object/process.

Whatever alternative has been chosen, an MRP has the following four responsibilities:

- supervise one or more measurement objects.
- map the measurement object identifier to the correct process identifier, thereby enabling communication with measurement objects.
- keep track of the available resources, mapping new measurement objects to the correct resource.
- ensure MESH that the user supplied supervision scheme is still working. That is, MESH only supervises the MRP (and the node it resides on); the MRP will supervise the individual measurement objects, reporting to MESH whenever one of them gets disabled (and order revival, if that is desired).

Should an MRP terminate, MESH will assume that the corresponding measurement objects will also terminate. This implies that MRPs have to be very robust, ie. they have to trap exits, even internally.

Events and Alarms

The alarms that may be sent to EVA are:

- `meshThresholdTriggered`. This alarm is sent whenever a threshold has been triggered. The alarm class is `qos`, and the severity `indeterminate`. Other fields of interest in the alarm are:

```
Field        Value
-----        -----
sender       mesh_server
cause        {upper_threshold_triggered, {value, number()}}  |
             {lower_threshold_triggered, {value, number()}}
extra        {{meas, MeasId}, {id, ThreshId}}
```

- meshTooManyTypes. This alarm is sent by the watchdog if the total number of registered measurement types exceeds the maximum number allowed. The alarm class is processing, and the severity warning. Other fields of interest in the alarm are:

```
Field        Value
-----        -----
sender       mesh_server
cause        {{allowed,number()},{currently,number()}}
extra        ""
```

- meshTooManyMeasurements. This alarm is sent by the watchdog if the total number of created measurement objects exceeds the maximum number allowed. The alarm class is processing, and the severity warning. Other fields of interest in the alarm are:

```
Field        Value
-----        -----
sender       mesh_server
cause        {{allowed,number()}, {currently,number()}}
extra        ""
```

- meshTypeCapacityExceeded. This alarm is sent when the number of measurement objects belonging to a certain measurement type exceeds the measurement type capacity. The alarm class is processing, and the severity warning. Other fields of interest in the alarm are:

```
Field        Value
-----        -----
sender       mesh_server
cause        {{type,TypeId}, {allowed,number()}, {currently,number()}}
extra        "Capacity decreased"
```

The events that may be sent to EVA are:

- meshTypeFailure. This event is sent whenever a measurement type MRP has terminated. The extra field in the event contains the following information: {MeasId, FailureReason, FailureTime}.
- meshMeasurementTerminated. This event is sent whenever the Measurement Handler has noticed that a measurement object has been disabled. The extra field in the event contains the following information: {MeasId, TypeId, TerminationReason, TerminationTime}.
- meshNodeUp. This event is sent whenever a node (where an MRP is residing), comes up again after having been down. The extra field in the event contains the following information: {NodeName, ConnectionTime}.
- meshNodeDown. This event is sent whenever a node (where an MRP is residing),goes down. The extra field in the event contains the following information: {NodeName, CrashTime}.

- `meshTypeUnconnected`. This event is sent when the status of an MRP is indeterminate, for example when the connection to the corresponding node has been lost, but no crash report has been received. The `extra` field in the event contains the following information:
  {`TypeId, nodedown, Time`}.
- `meshMeasurementUnconnected`. This event is sent when the status of a measurement object is indeterminate, for example when the connection to the corresponding node has been lost, but no crash report has been received. The `extra` field in the event contains the following information:
  {`MeasId, TypeId, nodedown, Time`}.
- `meshTypeConnected`. This event is sent when a previously unconnected MRP reconnects. For example, if the node where the MRP had resided is reconnected, and the MRP is found to still be present. The `extra` field in the event contains the following information:
  {`TypeId, nodeup, ConnectionTime`}.
- `meshMeasurementConnected`. This event is sent when a previously unconnected measurement object reconnects. For example, if the node where the measurement object resided is reconnected, and the measurement object is found to still be present. The `extra` field in the event contains the following information:
  {`MeasId, TypeId, nodeup, ConnectionTime`}.
- `meshMeasurementReport`. This event is sent whenever the Measurement Handler receives a measurement report from any measurement object. The `extra` field in the event contains the following information:
  {{`name,MeasId`}, {`value,number()`}, {`time,Time`}, {`info,MeasInfo`}}.

Adaptations

The MESH services are management protocol independent. However, for a manager to access the MESH services, a management protocol is required, ie. *adaptations* must be written, mapping MESH services to the desired protocol.

Exports

`create_tables(NodeList) -> Result`

Types:

- NodeList = [NodeName]
- NodeName = atom()
- Result = ok | {error, Reason}
- Reason = term()

Creates the Mnesia tables the MESH application needs, with disc and RAM replicas on every node specified in `NodeList`.
This function will only be called once, before starting the MESH application.

Note: it is important that the schema created is consistent with the specified list of nodes.

```
register_type(TypeId, Extra, InterfaceMod, NofInst) -> Result
register_type(TypeId, Extra, InterfaceMod, NofInst, AdminState) -> Result
```

Types:

- TypeId = atom()
- Extra = term()
- InterfaceMod = atom()
- NofInst = integer()
- AdminState = unlocked | shutting_down | locked
- Result = {registered, TypeId} | {reregistered, TypeId} | {error, Reason}
- Reason = term()

This function registers (or reregisters) a measurement type in the Measurement Handler. The main purpose of this function is to associate the `InterfaceMod` interface module with the `TypeId` identifier. (It should be noted that the `TypeId` identifier cannot be reused, since it solely identifies a measurement type, and therefore must be unique.)
The `Extra` argument is used for the benefit of the user, ie. it must contain all information describing the measurement type, but this information will only be used by the manager and never forwarded to any MRP or measurement object.
The registration function will also call the `init/1` function in the interface module, thereby enabling the user to perform necessary initializations. (The init function may, for example, start a MRP.)
The `NofInst` parameter sets the measurement type capacity, i.e., the number of measurement objects (belonging to the specified measurement type) that may be created.
The `AdminState` sets the initial administrative state of the the measurement type.

It is possible to re-register a measurement type, `provided that the same interface module is specified`. This allows changes in capacity, e.g., the number of measurement objects that may be created.
However, in the case of a capacity decrease, no measurement objects will be disabled, but an alarm will be sent to EVA.
When re-registration has been approved by MESH, the `init/1` function in the interface module will once again be called to restart possible existing MRPs.

Should the type capacity be exceeded, a `meshTypeCapacityExceeded` alarm will be sent to EVA. Once the number of existing measurement objects falls below the type capacity, the alarm will be cleared.

```
unregister_type(TypeId) -> Result
unregister_type(TypeId, StopArgs) -> Result
```

Types:

- TypeId = atom()
- StopArgs = [term()]
- Result = {unregistered, TypeId} | {error, Reason}
- Reason = term()

This function unregisters a measurement type in the Measurement Handler. Any existing measurement objects will be deleted, and the interface module will no longer be associated with the `TypeId` identifier. For each existing measurement object, the `delete_measurement/3` function in the measurement type interface will be called, with `StopArgs` as one of the arguments. Finally, the `terminate1` function in the measurement type interface will be called; this is to enable clean-up actions.
Note: The `StopArgs` are only passed on to the `delete_measurement` function, not to the `terminate` function!

`list_types() -> Result`

Types:

- Result = [TypeInfo] | {error, Reason}
- TypeInfo = {TypeId, [{extra,Extra}, {interface_mod,InterfaceMod}, {instances,CurrNofInst}, {max_instances,MaxNofInst}, {administrative_state,CurrAdminState}]}
- TypeId = atom()
- Extra = term()
- InterfaceMod = atom()
- CurrNofInst = integer()
- MaxNofInst = integer()
- CurrAdminState = unlocked | shutting_down | locked
- Reason = term()

This function lists all the measurement types currently registered in the Measurement Handler, including information about settings and the current state of each of the types.

`lock_type(TypeId) -> Result`
`lock_type(TypeId, StopArgs) -> Result`

Types:

- TypeId = atom()
- StopArgs = [term()]
- Result = {locked, TypeId} | {error, Reason}
- Reason = term()

This function locks the specified measurement type, ie. prevents further usage until it becomes unlocked. After locking the measurement type, it will be impossible to create any new measurement objects, and existing measurement objects will be deleted, using the `delete_measurement/3` function in the measurement type interface; the `StopArgs` will be passed on to this latter function, thereby enabling soft measurement object termination. (Unless otherwise stated, `StopArgs` will be set to the empty list.)

`unlock_type(TypeId) -> Result`

Types:

- TypeId = atom()
- Result = {unlocked, TypeId} | {error, Reason}
- Reason = term()

This function unlocks the specified measurement type. Once activated, it is possible to create measurement objects (provided that the type capacity isn't exceeded).

`shut_down_type(TypeId) -> Result`

Types:

- TypeId = atom()
- Result = {shutting_down, TypeId} | {locked, TypeId} | {error, Reason}
- Reason = term()

This function shuts down the specified measurement type. Once effective, no measurement objects can be created and measurement objects cannot be revived, once they have been disabled.
Existing measurement objects are allowed to continue to exist, but once all measurement objects (belonging to the specified type) have been disabled or deleted, the measurement type is automatically locked.

`create_measurement(MeasId, TypeId, Extra, ResId) -> Result`
`create_measurement(MeasId, TypeId, Extra, ResId, AdminState) -> Result`
`create_measurement(MeasId, TypeId, Extra, ResId, AdminState, StartArgs) -> Result`

Types:

- MeasId = atom()
- TypeId = atom()
- Extra = term()
- ResId = term()
- AdminState = started | stopped
- StartArgs = [term()]
- Result = {created, MeasId} | {error, Reason}

This function creates a new measurement object belonging to the specified measurement type, provided the type capacity isn't exceeded. The `create_measurement/5` function in the type interface will be called, with some of the supplied arguments, as a result of this function call. Please see below, and the section relating to `Interface Module Functions`.
The `MeasId` identifier, during the lifetime of the object, is used to uniquely identify the object. It follows that there can only be one measurement object with a unique measurement identifier.
The `TypeId` identifier tells the measurement type which interface module to use when creating the object and communicating with it; the `TypeId` identifier is the same as was given when the measurement type was registered.
The `Extra` argument is any additional information about the measurement object the user chooses to supply; this information is `not` forwarded to the measurement object itself.
The `ResId` argument is any term telling the new measurement object which resources to use; the interpretation of this term is solely a user issue!
The `AdminState` tells the original administrative state of the new measurement object, ie. whether it should be `started` or `stopped`. The default administrative state is `started`.
`StartArgs` is any list of terms the user wants to forward to the new measurement object as start arguments, setting the original state, eg. the sampling interval and the algorithm, to use when evaluating samples are collected. Unless specified by the user, `StartArgs` will be set to the empty list when forwarded to the measurement object.

Note: the `StartArgs` list has to have the same format in the functions `create_measurement`, `start_measurement` and `reset_measurement`, see the description of function `revive_measurement` for further information.

```
delete_measurement(MeasId) -> Result
delete_measurement(MeasId, StopArgs) -> Result
```

Types:

- MeasId = atom()
- StopArgs = [term()]
- Result = {deleted, MeasId} | {error, Reason}

This function deletes the specified measurement object. As a result of this function call, the `delete_measurement/3` function in the measurement type interface will be called, with some of the supplied arguments. Please see below and section `Interface Module Functions` for further information.
`MeasId` is the measurement identifier used to identify the measurement object.
`StopArgs` is any list of terms the user wants to forward to the measurement object, thereby enabling a soft termination; the default is the empty list.
Note: it is the responsibility of the user to ensure that all resources and applications used by the measurement object are terminated and/or freed in a controlled manner when an object is terminated.

```
measurement_terminated(MeasId, Reason) -> ok
```

Types:

- MeasId = atom()
- Reason = term()

This function is used to inform the Measurement Handler about terminated/disabled measurement objects. Since the Measurement Handler only supervises the Measurement Responsible Processes (MRPs), it is strictly necessary that each MRP reports to MESH about terminated/disabled measurement objects. The operation of the Measurement Handler is not guaranteed, should the MRPs neglect this reporting. Should the measurement object itself be an MRP, not supervising any other measurement objects, the Measurement Handler will notice automatically (since it is linked to each MRP) if the measurement object terminates, and no specific reporting is needed in this case. In all other cases the MRPs `must` report each measurement object termination to MESH, using this function.

```
list_measurements(TypeId) -> Result
```

Types:

- TypeId = atom()
- Result = [MeasInfo] | {error, Reason}
- MeasInfo = {MeasId, [{extra,Extra}, {resources,ResId}, {initial_arguments,StartArgs}, {operability_state,CurrOperState}, {administrative_state,CurrAdminState}]}
- MeasId = atom()
- Extra = term()
- ResId = term()
- StartArgs = [term()]
- CurrOperState = enabled | disabled
- CurrAdminState = started | stopped
- Reason = term()

This function lists all measurement objects belonging to the specified measurement type. The listing includes information about the resources used, the initial state (ie. the last known start arguments), the current operability state (ie. whether the measurement object is enabled or disabled), and the administrative state (ie. whether the measurement object is started or stopped).

Note: the `StartArgs` is the list of start arguments as given in any (ie. the latest) of the functions `create_measurement`, `start_measurement` and `reset_measurement`.

`revive_measurement(MeasId) -> Result`

Types:

- MeasId = atom()
- Result = {revived, MeasId} | {error, Reason}
- Reason = term()

This function tries to revive a disabled measurement object, using the last known settings, including set thresholds.

Please noted that the last known start arguments will be used when trying to revive a measurement object. These start arguments consist of the latest known `StartArgs` list; this list may be specified in any of these three function calls:

- create_measurement
- start_measurement
- reset_measurement

Therefore, the `StartArgs` list must have the same format in these three functions.

The last step in the revival procedure, resetting the measurement object, is ordered by MESH to ensure consistent states in the Measurement Handler and the measurement object.

`start_measurement(MeasId) -> Result`
`start_measurement(MeasId, StartArgs) -> Result`

Types:

- MeasId = atom()
- StartArgs = [term()]
- Result = {started, MeasId} | {error, Reason}
- Reason = term()

This function orders a specified measurement object to start running (ie. to enter a working state). As a result of this function call, the `start_measurement/3` function in the measurement type interface will be called, with some of the supplied arguments. Please see below and section `Interface Module Functions` for further information. No resetting takes place, with the exception of thresholds set: if previously triggered, they are restored to an un-triggered state. (This applies only to the Measurement Handler, if the user has implemented thresholds in the measurement object, it is their own responsibility to ensure they are set to a consistent state.)

`MeasId` is the measurement identifier used to identify the measurement object. `StartArgs` is any list of terms the user wants to forward to the measurement object, thereby changing its internal state. The default is set to the empty list.

Note: the StartArgs list must have the same format in the functions create_measurement, start_measurement and reset_measurement, see the description of function revive_measurement for further information.

```
stop_measurement(MeasId) -> Result
```

Types:

- MeasId = atom()
- Result = {stopped, MeasId} | {error, Reason}
- Reason = term()

This function orders the specified measurement object to stop running and become idle. As a result of this function call, the stop_measurement/2 function in the measurement type interface will be called, please see section Interface Module Functions. No resetting takes place in the Measurement Handler, and all thresholds set remain set. However, no measurement reports will be accepted from a stopped measurement object.

MeasId is the measurement identifier used to identify the measurement object.

```
reset_measurement(MeasId) -> Result
reset_measurement(MeasId, StartArgs) -> Result
```

Types:

- MeasId = atom()
- StartArgs = [term()]
- Result = {reset, MeasId} | {error, Reason}
- Reason = term()

This function orders the specified measurement object to reset its internal state, and resets in the Measurement Handler all measurement information stored about the measurement object. All thresholds set will remain set, but if previously triggered, they are restored to an un-triggered state.
As a result of this function call, the reset_measurement/2 function in the measurement type interface will be called, please see section Interface Module Functions.

MeasId is the measurement identifier used to identify the measurement object.
StartArgs is any list of terms the user wants to forward to the measurement object, thereby changing its internal state. Default is set to the empty list.
Note: the StartArgs list has to have the same format in the functions create_measurement, start_measurement and reset_measurement, see the description of function revive_measurement for further information.

```
measurement_report(MeasId, Value, TimeStamp) -> ok
measurement_report(MeasId, Value, TimeStamp, MeasInfo) -> ok
```

Types:

- MeasId = atom()
- Value = number()
- TimeStamp = term()
- MeasInfo = term()

This function is used to report a measurement value, obtained by any measurement object, to the Measurement Handler. The Measurement Handler will update the tidemarks and compare the received value to the thresholds set, and when required, forward the measurement report to EVA. The measurement value will also be stored (internally) in the Measurement Handler, until a new measurement report is received from the same measurement object.

`MeasId` is the measurement identifier used to identify the measurement object.
`Value` is the measurement value the measurement object has obtained.
`TimeStamp` is any (user specified) term describing the time and date the measurement value and/or report was obtained or sent.
`MeasInfo` is any (user specified) term describing the measurement report and/or providing extra information to the manager.

`get_measurement_report(MeasId) -> Result`

Types:

- MeasId = atom()
- Result = {MeasId, Value, TimeStamp, MeasInfo} | {error, Reason}
- Value = number()
- TimeStamp = term()
- MeasInfo = term()
- Reason = term

This function gets the most recently reported measurement value from a specified measurement object.

`MeasId` is the measurement identifier used to identify the measurement object.
`Value` is the measurement value reported by the measurement object.
`TimeStamp` is any (user specified) term describing the time and date the measurement value and/or report was obtained or sent.
`MeasInfo` is any (user specified) term describing the measurement report and/or providing extra information to the manager.

`set_upper_threshold(MeasId, ThreshId, Value) -> Result`
`set_upper_threshold(MeasId, ThreshId, Value, Status) -> Result`

Types:

- MeasId = atom()
- ThreshId = atom()
- Value = number() | {ValueHi, ValueLo}
- ValueHi = ValueLo = number()
- Status = enabled | disabled
- Result = {threshold_set, {MeasId,ThreshId}} | {error, Reason}
- Reason = term()

This function sets an upper threshold in the Measurement Handler. Thus, when a threshold has been exceeded an alarm will be triggered. The specified measurement object also receives the threshold information, using the measurement type interface function `set_upper_threshold/5`, to allow the user to implement the threshold functionality on their own. However, it should be noted that the `set_upper_threshold` function in the measurement type interface is purely optional. The Measurement Handler will not crash, should the function not be present, because, MESH will not check the return value.

MeasId is the measurement identifier used to identify the measurement object.
ThreshId is the threshold identifier, which is used together with the measurement identifier to uniquely identify a threshold. Therefore, two different measurement objects may use the same threshold identifier.
Value is a single number or a pair of numbers, specifying the threshold level. In the case of a pair of numbers, the first one is assumed to specify the level (value) any measurement value has to exceed in order for the threshold to be triggered. The second number is the value any measurement value has to fall below in order for a previously triggered threshold to be cleared (untriggered).
If only one value is given, the two levels are assumed to be the same.
Status specifies whether the threshold will be enabled or disabled when created. Only enabled thresholds can be triggered.

A triggered threshold will issue a meshThresholdTriggered alarm to EVA. When the threshold becomes untriggered, the alarm is cleared.

```
set_lower_threshold(MeasId, ThreshId, Value) -> Result
set_lower_threshold(MeasId, ThreshId, Value, Status) -> Result
```

Types:

- MeasId = atom()
- ThreshId = atom()
- Value = number() | {ValueLo, ValueHi}
- ValueHi = ValueLo = number()
- Status = enabled | disabled
- Result = {threshold_set, {MeasId,ThreshId}} | {error, Reason}
- Reason = term()

This function sets a lower threshold in the Measurement Handler. Therefore, an obtained measurement value must fall below the lower threshold in order for the threshold alarm to be triggered. The specified measurement object also receives the threshold information, using the measurement type interface function set_lower_threshold/5, should the user want to implement the threshold functionality on their own. However, it should be noted that the set_lower_threshold function in the measurement type interface is purely optional. The Measurement Handler will not crash, should the function not be present, because, MESH doesn't check the return value.

MeasId is the measurement identifier used to identify the measurement object.
ThreshId is the threshold identifier, which is used together with the measurement identifier to uniquely identify a threshold. Thus, two different measurement objects may use the same threshold identifier.
Value is a single number or a pair of numbers, specifying the threshold level. In the case of a pair of numbers, the first one is assumed to specify the level (value) any measurement value has to fall below in order for the threshold to be triggered. The second number is the value any measurement value has to exceed in order for a previously triggered threshold to be cleared (untriggered).
If only one value is given, the two levels are assumed to be the same.
Status specifies whether the threshold will be enabled or disabled when created. Only enabled thresholds can be triggered.

A triggered threshold will activate a meshThresholdTriggered alarm to EVA. When the threshold becomes untriggered, the alarm is cleared.

`remove_threshold(MeasId, ThreshId) -> Result`

Types:

- MeasId = atom()
- ThreshId = atom()
- Result = {threshold_removed, {MeasId,ThreshId}} | {error, Reason}
- Reason = term()

This function removes a specified threshold from the specified measurement object. The specified measurement object also receives the threshold removal order, via the measurement type interface function `remove_threshold/3`, should the user want to implement the threshold functionality on his own. However, it should be noted that the `remove_threshold` function in the measurement type interface is purely optional. The Measurement Handler will not crash, should the function not be present because, MESH doesn't check the return value.

`MeasId` is the measurement identifier used to identify the measurement object. `ThreshId` is the threshold identifier, which is used `together` with the measurement identifier to uniquely identify a threshold.

`remove_thresholds(MeasId) -> Result`

Types:

- MeasId = atom()
- Result = {thresholds_removed, MeasId} | {error, Reason}
- Reason = term()

This function removes all existing thresholds from the specified measurement object. The specified measurement object also receives the threshold removal order, via (repetitive calls to) the measurement type interface function `remove_threshold/3`, should the user want to implement the threshold functionality on their own. However, it should be noted that the `remove_threshold` function in the measurement type interface is purely optional. The Measurement Handler will not crash, should the function not be present because, MESH won't even check the return value.

`MeasId` is the measurement identifier used to identify the measurement object.

`list_thresholds(MeasId) -> Result`

Types:

- MeasId = atom()
- Result = {MeasId, {upper_thresholds,[ThreshInfo]}, {lower_thresholds,[ThreshInfo]}} | {error, Reason}
- ThreshInfo = {ThreshId, Status, Value}
- ThreshId = atom()
- Status = enabled | disabled
- Value = number() | {number(), number()}
- Reason = term()

This function lists all existing thresholds set for the specified measurement object. The thresholds are divided into the two categories `upper thresholds` and `lower thresholds`. For each threshold, the current status is specified (`enabled` or `disabled`), as well as the threshold value(s) set.

`MeasId` is the measurement identifier used to identify the measurement object.

`enable_threshold(MeasId, ThreshId) -> Result`

Types:

- MeasId = atom()
- ThreshId = atom()
- Result = {threshold_enabled, {MeasId,ThreshId}} | {error, Reason}
- Reason = term()

This function enables the specified threshold. Only enabled thresholds may be triggered. The specified measurement object also receives the enable threshold order, via the measurement type interface function `enable_threshold/3`, should the user want to implement the threshold functionality on his own. However, it should be noted that the `enable_threshold` function in the measurement type interface is purely optional. The Measurement Handler will not crash, should the function not be present because, MESH will not even check the return value.

`MeasId` is the measurement identifier used to identify the measurement object. `ThreshId` is the threshold identifier, which is used `together` with the measurement identifier to uniquely identify a threshold.

`disable_threshold(MeasId, ThreshId) -> Result`

Types:

- MeasId = atom()
- ThreshId = atom()
- Result = {threshold_disabled, {MeasId,ThreshId}} | {error, Reason}
- Reason = term()

This function disables a specified threshold. Disabled thresholds cannot be triggered. The specified measurement object also receives the disable threshold order, via the measurement type interface function `disable_threshold/3`, should the user want to implement the threshold functionality on his own. However, it should be noted that the `disable_threshold` function in the measurement type interface is purely optional The Measurement Handler will not crash, should the function not be present because, MESH will not even check the return value.

`MeasId` is the measurement identifier used to identify the measurement object. `ThreshId` is the threshold identifier, which is used *together* with the measurement identifier to uniquely identify a threshold.

`report_tidemarks(MeasId) -> Result`

Types:

- MeasId = atom()
- Result = {MeasId, TypeId, MaxTideMark, MinTideMark} | {error, Reason}
- TypeId = atom()
- MaxTideMark = {max_tidemark, [{current,CurrValue}, {previous,PrevValue}, {reset,ResetTime}]}
- MinTideMark = {min_tidemark, [{current,CurrValue}, {previous,PrevValue}, {reset,ResetTime}]}
- CurrValue = PrevValue = number()
- ResetTime = {Date, Time}
- Date = {Year, Month, Day}

- Time = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = int()
- Reason = term()

This function reports the current values of the maximum and minimum tide-marks associated with the specified measurement object.
Each tide-mark contains the maximum (or minimum) value reached since the last time the tide-mark was reset, and also the maximum (or minimum) value reached in the period prior to the last reset.

`MeasId` is the measurement identifier used to identify the measurement object.
`TypeId` is the measurement type identifier.

Note: the time is given using the Universal Coordinated Time time zone (sometimes denoted Greenwich Mean Time or GMT).

`reset_tidemarks(MeasId) -> Result`

Types:

- MeasId = atom()
- Result = {tidemarks_reset, MeasId} | {error, Reason}
- Reason = term()

This function resets the maximum and minimum tide-marks associated with a specified measurement object. Resetting consists of storing the current value as the previous current value, replacing the current value with the atom 'undefined', and noting the reset time.

`MeasId` is the measurement identifier used to identify the measurement object.

`watchdog_setup(NofTypes, NofMeas) -> Result`

Types:

- NofTypes = NofMeas = int()
- Result = ok | {error, Reason}
- Reason = term()

This functions handles the set-up of the Measurement Handler watchdog. The purpose of the watchdog is to keep track of the `total` number of measurement types registered, as well as the total number of measurement objects created. Should the number of types exceed the number specified using this function, the alarm `meshTooManyTypes` is sent to EVA, however, the alarm `meshTooManyMeasurements` is sent if the number of allowed measurement objects is exceeded.
It should be noted that the alarms issued are just warnings and it is still possible to register new measurement types, and also to create new measurement objects (provided that the associated measurement type capacity isn't exceeded).

Functions used to control the various logs

MESH uses three logs:

- `alarms` are logged in the `mesh_alarms` log.
- `events` are logged in the `mesh_events` log.
- `measurement reports` are logged in the `mesh_measurements` log.

The user may freely choose to replace the filter function used by each log, using a user defined filter function instead, or to restore the original filter function.

Exports

`set_alarm_filter(Func) -> Result`

Types:

- Func = {Mod, Fcn, Args}
- Mod = Fcn = atom()
- Args = [term()]
- Result = ok | {error, Reason}
- Reason = term()

This function replaces the default filter with the user specified. (The default filter ensures that only alarms are recorded in this log, but doesn't perform any other action.)

`set_event_filter(Func) -> Result`

Types:

- Func = {Mod, Fcn, Args}
- Mod = Fcn = atom()
- Args = [term()]
- Result = ok | {error, Reason}
- Reason = term()

This function replaces the default filter with a user specified filter. The default filter ensures that only events, with the exception of measurement report events, are logged in this log, but won't perform any other action.

`set_measurement_filter(Func) -> Result`

Types:

- Func = {Mod, Fcn, Args}
- Mod = Fcn = atom()
- Args = [term()]
- Result = ok | {error, Reason}
- Reason = term()

This function replaces the default filter with the user specified filter. The default filter ensures that only measurement report events are recorded in this log, but won't perform any other action.

`reset_alarm_filter() -> Result`

Types:

- Result = ok | {error, Reason}
- Reason = term()

This function restores the filter used to the default one.

`reset_event_filter() -> Result`

Types:

- Result = ok | {error, Reason}
- Reason = term()

This function restores the filter used to the default one.

`reset_measurement_filter() -> Result`

Types:

- Result = ok | {error, Reason}
- Reason = term()

This function restores the filter used to the default one.

Interface Module Functions

The Measurement Handler application calls a number of functions in each `measurement type interface` module. Only if explicitly stated below is it optional to export a specific function.
The absence of a required function won't cause the Measurement Handler to crash, but it will severely limit the possibilities when handling the corresponding measurement types and measurement objects.

Exports

`InterfaceMod:init(TypeId) -> Result`

Types:

- TypeId = atom()
- Result = MRP | {error, Reason}
- MRP = pid() | undefined
- Reason = term()

This function takes care of the user application specific initialization. Should there be an MRP responsible for a number of measurement objects, it has to be started by this function, in which case the PID of the MRP will have the return value of the function. If no MRP is started, the atom 'undefined' will be returned.

`InterfaceMod:terminate(MRP) -> Result`

Types:

- MRP = pid() | undefined
- Result = ok | {error, Reason}
- Reason = term()

This function is responsible for user specific termination and clean-up. If resources have to be freed, and/or applications terminated, this function has to handle it.

The `MRP` argument is the identifier once returned from the `InterfaceMod:init` function.

`InterfaceMod:create_measurement(MRP1, TypeId, MeasId, ResId, StartArgs) -> Result`

Types:

- MRP1 = pid() | undefined
- TypeId = atom()
- MeasId = atom()
- ResId = term()
- StartArgs = [term()]
- Result = MRP2 | {error, Reason}
- MRP2 = pid()
- Reason = term()

This function handles the actual creation of a measurement object. It is the user's responsibility to make sure that the created object uses the correct resources, and performs measurements as intended.

The `MRP1` argument is the identifier once returned from the `InterfaceMod:init` function. This identifier may be a pid, or it may be the atom 'undefined'. However, the return value of the function has to be a PID, namely the process identifier of the process supervising the measurement object (which actually may be the measurement object itself).

`InterfaceMod:start_measurement(MRP, MeasId, StartArgs) -> Result`

Types:

- MRP = pid()
- MeasId = atom()
- StartArgs = [term()]
- Result = ok | {error, Reason}
- Reason = term()

This function handles a measurement object start.

The MRP argument is the identifier returned from the InterfaceMod:create_measurement function.

`InterfaceMod:stop_measurement(MRP, MeasId) -> Result`

Types:

- MRP = pid()
- MeasId = atom()
- Result = ok | {error, Reason}
- Reason = term()

This function handles a measurement object stop.

The MRP argument is the identifier returned from the InterfaceMod:create_measurement function.

`InterfaceMod:reset_measurement(MRP, MeasId, StartArgs) -> Result`

Types:

- MRP = pid()
- MeasId = atom()
- StartArgs = [term()]
- Result = ok | {error, Reason}
- Reason = term()

This function is responsible for the resetting of a measurement object.

The MRP argument is the identifier returned from the InterfaceMod:create_measurement function.

`InterfaceMod:delete_measurement(MRP, MeasId, StopArgs) -> Result`

Types:

- MRP = pid()
- MeasId = atom()
- StopArgs = [term()]
- Result = ok | {error, Reason}
- Reason = term()

This function deletes the specified measurement object.

The MRP argument is the identifier returned from the InterfaceMod:create_measurement function.

`InterfaceMod:set_upper_threshold(MRP, MeasId, ThreshId, Value, Status) -> Result`

Types:

- MRP = pid()

- MeasId = atom()
- ThreshId = atom()
- Value = number() | {ValueHi, ValueLo}
- ValueHi = ValueLo = number()
- Status = enabled | disabled
- Result = ok | {error, Reason}
- Reason = term()

This function sets an upper threshold in a measurement object.

The `MRP` argument is the identifier returned from the `InterfaceMod:create_measurement` function.

Note: the existence of this function is optional!

`InterfaceMod:set_lower_threshold(MRP, MeasId, ThreshId, Value, Status) -> Result`

Types:

- MRP = pid()
- MeasId = atom()
- ThreshId = atom()
- Value = number() | {ValueLo, ValueHi}
- ValueLo = ValueHi = number()
- Status = enabled | disabled
- Result = ok | {error, Reason}
- Reason = term()

This function sets a lower threshold in a measurement object.

The `MRP` argument is the identifier returned from the `InterfaceMod:create_measurement` function.

Note: the existence of this function is optional.

`InterfaceMod:remove_threshold(MRP, MeasId, ThreshId) -> Result`

Types:

- MRP = pid()
- MeasId = atom()
- ThreshId = atom()
- Result = ok | {error, Reason}
- Reason = term()

This function removes a specified threshold in a measurement object.

The `MRP` argument is the identifier returned from the `InterfaceMod:create_measurement` function.

Note: the existence of this function is optional.

`InterfaceMod:enable_threshold(MRP, MeasId, ThreshId) -> Result`

Types:

- MRP = pid()
- MeasId = atom()
- ThreshId = atom()

- Result = ok | {error, Reason}
- Reason = term()

This function enables a specified threshold in a measurement object.

The MRP argument is the identifier returned from the `InterfaceMod:create_measurement` function.

Note: the existence of this function is optional.

```
InterfaceMod:disable_threshold(MRP, MeasId, ThreshId) -> Result
```

Types:

- MRP = pid()
- MeasId = atom()
- ThreshId = atom()
- Result = ok | {error, Reason}
- Reason = term()

This function disables a specified threshold in a measurement object.

The MRP argument is the identifier returned from the `InterfaceMod:create_measurement` function.

Note: the existence of this function is optional.

SEE ALSO

mesh_lib(3), mesh_snmp(3), application(3), disk_log(3), eva(3), eva_log(3), eva_sup(3), mnesia(3)

mesh_lib (Module)

This module contains the Measurement Handler library functions. The purpose of the library is to provide the user with commonly used statistical functions.

Exports

`sum(SampleList) -> Result`

Types:

- SampleList = [number()]
- Result = SampleSum | {error, Reason}
- SampleSum = number()
- Reason = term()

This function adds up a list of measurement samples.

`sum_and_squaresum(SampleList) -> Result`

Types:

- SampleList = [number()]
- Result = {SampleSum, SampleSquareSum} | {error, Reason}
- SampleSum = SampleSquareSum = number()
- Reason = term()

This function adds up a list of measurement samples, and also sums the square of each sample.

`sample_mean(SampleList) -> Result`

Types:

- SampleList = [number()]
- Result = EstimatedSampleMean | {error, Reason}
- EstimatedSampleMean = number()
- Reason = term()

Suppose that [X1, X2, X3, ..., Xn] are random variables (observations) with finite population mean u.
The sample mean estimation m(n) = (X1 + X2 + X3 + ... +Xn) / n

`sample_variance(SampleList) -> Result`

Types:

- SampleList = [number()]
- Result = EstimatedSampleVariance | {error, Reason}
- EstimatedSampleVariance = number()
- Reason = term()

Suppose that [X1, X2, X3, ..., Xn] are random variables (observations) with finite population mean u and finite population variance V, and sample mean estimate m(n). The sample variance estimation S^2(n) = ((X1 - m(n))^2 + (X2 - m(n))^2 + ... + (Xn - m(n))^2) / (n - 1) =
= (X1^2 + X2^2 + ... + Xn^2 - (n * m(n)^2)) / (n - 1)

`sample_mean_and_variance(SampleList) -> Result`

Types:

- SampleList = [number()]
- Result = {EstimatedSampleMean, EstimatedSampleVariance} | {error, Reason}
- EstimatedSampleMean = EstimatedSampleVariance = number()
- Reason = term()

Suppose that [X1, X2, X3, ..., Xn] are random variables (observations) with a finite population mean u with a finite population variance V, and a sample mean estimate m(n). The sample mean estimation m(n) = (X1 + X2 + X3 + ... +Xn) / n The sample variance estimation S^2(n) = ((X1 - m(n))^2 + (X2 - m(n))^2 + ... + (Xn - m(n))^2) / (n - 1) =
= (X1^2 + X2^2 + ... + Xn^2 - (n * m(n)^2)) / (n - 1)

`mean_variance(MeanList) -> Result`

Types:

- MeanList = [EstimatedMeans]
- EstimatedMeans = number()
- Result = MeanVariance | {error, Reason}
- MeanVariance = number()
- Reason = term()

Suppose that [X1, X2, X3, ..., Xn] are *independent* random variables with a finite population mean u, a finite population variance V, and a sample mean estimate m(n). The variance of the estimated mean may be equated through the formula Var(m(n)) = S^2(n) / n NOTE: This formula is valid only if all Xi's are independent (uncorrelated)!!! This is normally not the case in simulations.

`ewma_mean(Xnew, Wn, GP, MTP) -> Result`

Types:

- Xnew = Wn = GP = MTP = number()
- Result = EstimatedMean | {error, Reason
- EstimatedMean = number()
- Reason = term()

This function computes the mean of a number of samples using the Exponentially Weighted Moving Average technique. Suppose that [X1, X2, X3, ..., Xn] are random variables (observations) with finite population mean u. Assume we have previously equated a mean value estimate Wn (where W0 may have been simply estimated). Let GP denote the granularity period, ie., the time elapsed between any two successive sample measurements, and let MTP denote the moving time period, ie., the time within which samples are considered. (For example, let GP be 5 ms, and MTP 1 s, which means that the EWMA mean will be based on 200 samples.) When we receive a new sample Xnew, the new estimate of the mean will be Wnew = f * Xnew + (1 - f) * Wn, where f = 2 * GP / (GP + MTP)

`ewma_variance(Xnew, Wnew, Sn, GP, SMTP) -> Result`

Types:

- Xnew = Wnew = Sn = GP = SMTP = number()
- Result = EstimatedVariance | {error, Reason
- EstimatedVariance = number()
- Reason = term()

This function computes the variance of a number of samples using the Exponentially Weighted Moving Average technique. Suppose that [X1, X2, X3, ..., Xn] are random variables (observations) with a finite population mean u. Also assume we have previously computed a variance value estimate Sn (where S0 may have been simply calculated). Let GP denote the granularity period, ie., the time elapsed between any two successive sample measurements. Also, let SMTP denote the second moving time period, ie., the effective time interval over which values are scanned to calculate an estimate of the variance. When we receive a new sample Xnew, the new estimate of the mean will be Snew = g * (Xnew - Wnew)^2 + (1 - g) * Sn, where g = 2 * GP / (GP + SMTP) NOTE: the bias can be shown to be u = 2*(1-f)^2/(2-f), times the variance of Xnew - Wnew (where f is taken from the ewma_mean formula). This may be used to reduce the bias in the calculations, using the formula S'n = Sn / u. The manager may decide whether to reduce the bias or ignore it.

`uwma_mean(Xnew, SX, Xold, N) -> Result`

Types:

- Xnew = SX = Xold = N = number()
- Result = {EstimatedMean, SampleSum} | {error, Reason
- EstimatedMean = SampleSum = number()
- Reason = term()

This function computes the variance of a number of samples using the Uniformly Weighted Moving Average technique. Suppose that [X1, X2, X3, ..., Xn] are random variables (observations) with finite population mean u, and sample sum SX. When we receive a new sample Xnew, the new estimate of the mean will be Wnew = Xnew + (SX - Xold) / N, where Xold is the oldest sample used in the calculation of SX (ie., the sample that will be replaced by Xnew when calculating Wnew), and N is the number of samples the calculation of Wnew is based on.

`uwma_variance(Xnew, SX, SqSX, Xold, N) -> Result`

Types:

- Xnew = SX = SqSX = Xold = N = number()

- Result = {EstimatedVariance, SampleSum, SampleSquareSum} | {error, Reason
- EstimatedVariance = SampleSum = SampleSquareSum = number()
- Reason = term()

This function computes the variance of a number of samples using the Uniformaly Weighted Moving Average technique. Suppose that [X1, X2, X3, ..., Xn] are random variables (observations) with finite population mean u and finite population variance V, and sample sum SX, and sample square sum SqSX, When we receive a new sample Xnew, the new estimate of the variance will be Snew = ((Xnew^2 + SqSX - Xold^2) - (Xnew + SX - Xold)^2 / N) / (N - 1), where Xold is the oldest sample used in the calculation of SX (ie., the sample that will be replaced by Xnew when calculating Snew), and N is the number of samples the calculation of Snew is based on.

SEE ALSO

mesh(3)

mesh_snmp (Module)

This module implements an SNMP adaptation to basic MESH. The MIB implemented by this adaptation is OTP-MESH-MIB, which is located in the directory `mibs` in the distribution.

The MESH SNMP adaptation is started if the environment variable `snmp_adapted` is set to `true` when the MESH application is started.

The MESH SNMP adaptation requires that the EVA SNMP adaptation is started (and still running). The events and alarms generated by MESH is converted to traps using the API provided by the EVA SNMP adaptation. The traps are normally sent to community “standard trap”; the user has the possibility to change this.

Exports

`register_alarms(Community) -> void()`

Types:

- Community = string()

This function registers all alarms, used by the MESH application, in the EVA SNMP adaptation. This means that each alarm generated will cause a trap to be sent to the `Community`.

If this function isn’t called by the user, the traps are automatically sent to community “standard trap”.

`register_events(Community) -> void()`

Types:

- Community = string()

This function registers all events, used by the MESH application, in the EVA SNMP adaptation. This means that each event generated will cause a trap to be sent to the `Community`.

If this function isn’t called by the user, the traps are automatically sent to community “standard trap”.

SEE ALSO

mesh(3), mesh_lib(3), eva_snmp_adatation(3), snmp(3)

List of Figures

Glossary

Measurement Object

A user installed and configured application or process collecting and filtering measurement samples provided by one or more resource objects.
Local for chapter 1.

Measurement Responsible Process

A gateway between the measurement type interface and the individual measurement objects. A Measurement Responsible Process supervises one or more measurement objects, maps the measurement object identifier to the correct process identifier, thereby enabling communication with measurement objects. It also keeps track of the available resources, mapping new measurement objects to the correct resource and by its existence ensures MESH that the user supplied supervision scheme is still working. MESH only supervises the MRP (and the node it resides on), whereas the MRP has to supervise every individual measurement object, reporting to MESH whenever one becomes disable, and order revival, if required.
Local for chapter 1.

Measurement Type

A container, holding all the code needed when dealing with the associated measurement objects. Specifically, the measurement type will provide an interface, which is used by the Measurement Handler to create, delete, and control the measurement objects.
Local for chapter 1.

MESH adaptation

Provides a mapping from the generic MESH support to a specific management protocol.
Local for chapter 1.

Resource Object

A user or operating system installed and configured application or device producing measurement samples.
Local for chapter 1.

Index

Modules are typed in *this way*.
Functions are typed in `this way`.