# Event and Alarm handling Application(EVA)

**version 2.0**

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

# Contents

# Chapter 1

# EVA User's Guide

The Event and Alarm Handling application, *EVA* is an operation and maintenance application in the fault management area. It contains support for applications to send events and alarms and create logs. It provides managers functions for controlling the events and alarms and controlling the logs.

EVA is a management protocol independent application that needs protocol adaptations to communicate with a remote manager. Currently, one such adaptation for SNMP is included in EVA.

## 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 Fault Management application EVA. EVA consists of support for Event and Alarm Handling and support for generic Log Control.

EVA is a modular application that consists of two management protocol independent services. It contains also rules and functions for defining protocol adaptations for EVA. One such adaptation is included for SNMP. It consists of SNMP MIBs and implementation of these MIBs.

EVA uses the applications Mnesia and SASL.

### 1.1.1 Architecture

EVA is a subapplication that can be included into another application. It is designed to work as a distributed application, which means that it always executes on one node, with other nodes as standby nodes. EVA should run on the same node as other operation and maintenance applications, specifically the management protocol termination application, to minimize internal network traffic.

EVA is designed to be protocol independent, and may be used with different management protocols. For each such protocol, an *EVA adaptation* must be written. For example, we have defined an SNMP adaptation, and more adaptations may be defined in the future, e.g. for HTTP, CORBA or any proprietary protocol (e.g. plain Erlang).

The event and alarm support can run in two different modes, in *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 server running on one node in the network of Erlang nodes that the system consists of.

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

## 1.2 Services

There are two management protocol independent EVA services provided, the basic Event and Alarm service and the Log Control service. The basic EVA service provides clients with an API for registering and sending events and alarms. The Log control service provides a mechanism for control of generic logs. Also included is a specialization of the generic log function for logging of events and alarms.

Each service provides client functions that can be used from applications in the system to, for example, send alarms. There is also an API that management applications can use to monitor and control the system. This API can be extended for specific management protocols, such as SNMP or CORBA.

### 1.2.1 Basic Event and Alarm Service

This service contains functions for the client API to EVA. EVA is a distributed global application, which means that clients can access the EVA functionality from any node.

Clients can register and send events and alarms. Management applications can subscribe to event and alarms, and control the treatment of them.

An *event* is a notification sent from the *NE* to a management application. An event is uniquely identified by its name. A special form of an event is an *alarm*. An alarm represents a fault in the system that needs to be reported to the manager. An example of an alarm could be `equipment_on_fire`. When an alarm is sent, it becomes active, and is stored in an *active alarm list*. When the application that sent the alarm notices that the fault that caused the alarm is not valid anymore, it *clears* the alarm. When an alarm is cleared, the alarm is deleted from the active alarm list, and an `clear_alarm` event is generated by EVA. Each fault may give rise to several alarms, maybe with different severities. There can however only be one active alarm for each fault at the same time. For example, associated with disk space usage may be two alarms, `disk_80_percent_filled` and `disk_90_percent_filled`. These two alarms represents the same fault, but only one of them can be active at the same time. An active alarm is identified by its *fault_id*. In contrast to alarms, ordinary events do not represent faults, and are not stored as the alarms in the active alarm list.

The basic EVA server is a global server to which all events and alarms are sent. The server updates its tables (e.g. the active alarm list), and sends the event or alarm to the `alarm_handler` process that runs on the same node as the global server. `alarm_handler` is a `gen_event` process defined in the SASL application.

Before a client can send an event or alarm, the name of the event must be registered in EVA. To register an event, a client calls `register_event/2`. The parameters of this function are the name of the event and whether the event should be logged by default or not. A manager can decide to change this value later. To register an alarm, a client calls `register_alarm/4`. The parameters of this function are the name and logging parameters as for events, and the class and default severity of the alarm.

EVA stores the definitions of events and alarms in the Mnesia tables `eventTable` and `alarmTable` respectively. Since an alarm is a special form of an event, each alarm is present in both of these tables. The active alarm list is stored in the Mnesia table `alarm`. The records for all these tables are defined in the header file `eva.hrl`, available in the `include` directory in the distribution.

### Event Definition Table

All registered events are stored in the `eventTable`. It has the following attributes:

- `name`
- `log`
- `generated`

The event is uniquely identified by its `name`, which is an atom.

The `log` attribute is a boolean flag that tells whether this event should be stored in some log when it is generated or not. This attribute is writable.

The `generated` attribute is a counter that counts how many times the event has been generated.

### Alarm Definition Table

The `alarmTable` extends the `eventTable`, and has the following attributes:

- `name`
- `class`
- `severity`

The alarm is uniquely identified by its `name`, which is an atom. Note that each alarm is present in the `eventTable` as well.

The `class` attribute categorizes the alarm, and is defined when the alarm is registered. It is as defined in X.733, ITU Alarm Reporting Function:

- `communications`. An alarm of this class is principally associated with the procedures or processes required to convey information from one point to another.
- `qos`. An alarm of this class is principally associated with a degradation in the quality of service.
- `processing`. An alarm of this class is principally associated with a software or processing fault.
- `equipment`. An alarm of this class is principally associated with an equipment fault.
- `environmental`. An alarm of this class is principally associated with a condition relating to an enclosure in with equipment resides.

The `severity` parameter defines five severity levels, which provide an indication of how it is perceived that the capability of the managed object has been affected. Those severity levels which represent service affecting conditions ordered from most severe to least severe are `critical`, `major`, `minor` and `warning`. The levels used are as defined in X.733, ITU Alarm Reporting Function:

- `indeterminate`. The Indeterminate severity level indicates that the severity level cannot be determined.
- `critical`. The Critical severity level indicates that a service affecting condition has occurred and an immediate corrective action is required. Such a severity can be reported, for example, when a managed object becomes totally out of service and its capability must be restored.
- `major`. The Major severity level indicates that a service affecting condition has developed and an urgent corrective action is required. Such a severity can be reported, for example, when there is a severe degradation in the capability of the managed object and its full capability must be restored.

- `minor`. The Minor severity level indicates the existence of a non-service affecting fault condition and that corrective action should be taken in order to prevent a more serious (for example, service affecting) fault. Such a severity can be reported, for example, when the detected alarm condition is not currently degrading the capacity of the managed object.
- `warning`. The Warning severity level indicates the detection of a potential or impending service affecting fault, before any significant effects have been felt. Action should be taken to further diagnose (if necessary) and correct the problem in order to prevent it from becoming a more serious service affecting fault.

When an alarm is cleared, a `clear_alarm` event is generated. This event clears the alarm with the `fault_id` contained in the event. It is not required that the clearing of previously reported alarms are reported. Therefore, a managing system cannot assume that the absence of an `clear_alarm` event for a fault means that the condition that caused the generation of previous alarms is still present. Managed object definers shall state if, and under which conditions, the `clear_alarm` event is used.

### Active Alarm List

The active alarm list is stored in the ordered Mnesia table `alarm`. The corresponding record is sent to the `alarm_handler` when an alarm is sent. It has the following read-only attributes:

- `index`
- `fault_id`
- `name`
- `sender`
- `cause`
- `severity`
- `time`
- `extra`

A row in the active alarm list is uniquely identified by its `fault_id`. However, to make the table ordered, the alarms uses the integer `index` as a key into the table. For each new alarm, EVA allocates a new `index` that is greater than the `index` of all other active alarms.

The `name` is the name of the corresponding alarm type, defined in `alarmTable`.

`sender` is a term that uniquely identifies the resource that generated the alarm.

`cause` describes the probable cause of the alarm.

`severity` is the perceived severity of the alarm.

`time` is the UTC time the alarm was generated.

`extra` is any extra information describing the alarm.

### Event

When an event is generated, the `event` record is sent to `alarm_handler`. It has the following attributes:

- `name`
- `sender`
- `time`
- `extra`

The `name` is the name of the corresponding event type, defined in `eventTable`.

`sender` is a term that uniquely identifies the resource that generated the event.

`time` is the UTC time the event was generated.

`extra` is any extra information describing the event.

### Example

As an example of how to register and send events and alarms, consider the following code:

```
%%%----------------------------------------------------------------------
%%% Resource code
%%%----------------------------------------------------------------------
reg() ->
    eva:register_event(boardRemoved, true),
    eva:register_event(boardInserted, false),
    eva:register_alarm(boardFailure, true, equipment, minor).

remove_board(No) ->
    eva:send_event(boardRemoved, {board, No}, []).

insert_board(No, BoardName, BoardType) ->
    eva:send_event(boardInserted, {board, No}, {BoardName, BoardType}).

board_on_fire(No) ->
    FaultId = eva:get_fault_id(),
    %% Cause = fire, ExtraParams = []
    eva:send_alarm(boardFailure, FaultId, {board, No}, fire, []),
    FaultId.
```

Two events and one alarm is defined. Board removal is an event that is logged by default, and board insertion is an event that is not logged by default. The alarm `equipmentFailure` is a minor alarm that is logged by default.

When the application detects that board `N` is on fire, `board_on_fire(N)` is called. This function is responsible for sending the alarm. It gets a new fault identifier for the fault, and calls `eva:send_alarm/5`, pointing out the faulty board (`N`), and suggests that the probable cause for the equipment trouble is `fire`.

The `board_on_fire` function returns the fault identifier for the new alarm. This fault identifier can be used at a later time in a call to `eva:clear_alarm(FaultId)` to clear the alarm.

### 1.2.2 Log Control Service

The Log Control service contains functions for monitoring logs, and functions for transferring logs to remote hosts, e.g. management stations. The main purpose of the Log Control service is to provide one entity through which all logs in the system can be controlled by a management station. Regardless of the type log, all logs are controlled in a similiar fashion.

Clients can register their logs in the log server. Management applications can control the logs, and transfer the logs to a remote host.

#### Log Monitoring

This service uses a *log server* that monitors all logs in the system. Each log uses the standard module `disk_log` for the actual logging.

Each log has an administrative and an operational status, that both can be either `up` or `down`. If the operational status is `up`, the log is working, and if it is `down`, the log does not work. The administrative status is writable, and reflects the desired operational status. Normally they are both the same. If the administrative status is set to `up`, the operational status will be `up` as well. However, if the log for some reason does not work, e.g. if the disk partition is full, the operational status will be `down`. When the operational status is down, no events are logged in the log.

**Alarms** The `Tlog` service defines two EVA alarms; `log_file_error` and `log_wrap_too_often`.

- `log_file_error`. This alarm is generated if a file error occurs when an item is logged. Default severity is `critical`. The cause for this alarm can be any `Reason` as returned from `file:write` in case of error. The alarm is cleared if the file system starts working again. For example, the alarm can be generated if the partition is full, and cleared when space is available.
- `log_wrap_too_often`. This alarm is generated when the log wraps more often than the wrap time. Default severity is `major`. The cause for this alarm is undefined. The alarm is cleared if the log wraps within the wrap time, the next time it wraps.

**Example** The following is an example of code that creates a log to be controlled by the generic Log Control function:

```
start() ->
    disk_log:open([{name, "ex_log"},
                   {file, "ex_log/ex_log.LOG"},
                   {type, wrap},
                   {size, {10000, 4}}]),
    log:open("ex_log", ex_log_type, 3600).

test() ->
    %% Log an item
    disk_log:log("ex_log", {1, "log this"}),

    %% Set the administrative status of the log to 'down'
    log:set_admin_status("ex_log", down),

    %% Try  to log - this one won't be logged
    disk_log:log("ex_log", {2, "won't be logged"}),

    Logs1 = log:get_logs(),
```

```
    %% Set the administrative status of the log to 'up'
    log:set_admin_status("ex_log", up),

    %% Log an item
    disk_log:log("ex_log", {3, "log this"}),

    Logged = disk_log:chunk("ex_log", start),
    {Logs1, Logged}.
```

#### Log Transfer

It is possible to transfer a log to a remote host. When the log is transferred, the log may be filtered, and the log records may be formatted.

As the logs are implemented as `disk_log` logs, each log consists of several log files. When the log is transferred, it is written to one single file on the remote host. When `disk_log` is used, the log records are normally not formatted when they are stored in the log, in order to increase log performance. However, a manager will probably need the log formatted in a human readable format. Thus, when the log is being transferred, each log record may be formatted in a log specifc way. Of course, to further increase performance, the log can be transferred as is, and leave it to the managar to format the log off-line.

### 1.2.3 EVA Log Service

The EVA log service uses the generic Log Control service to implement log functionality for events and alarms defined in EVA.

In the rest of this description, the term *event* refers to both events and alarms as defined in EVA.

This log functionality supports logging of events from EVA. It uses the module `disk_log` for logging of events. There can be several event logs active at the same time. It is possible to create new event logs dynamically, either from within an application, or from a management system. Each log uses a filter function to decide whether an event should be stored in the log or not.

There is a concept of a default log. The default log is used to log any event that has the `log` flag in `eventTable` set to `true`, but no log is currently able to store the event (or there is no other log defined to log the event). The usage of the default log is optional.

For example, suppose that we want to define an alarm log, that logs all alarms in the system. We can do this with the following code:

```
-module(alarm_log).
-export([alarm_filter/1, make_alarm_log/0]).

alarm_filter(Item) when record(alarm, Item) -> true;
alarm_filter(_) -> false.

make_alarm_log() ->
  disk_log:open([{name, "alarm_log"},
                 {format, internal},
                 {type, wrap},
                 {size, {10000, 10}}]),
  eva_log:open("alarm_log", {alarm_log, alarm_filter, []}, 36000).
```

If we set the administrative status of this log to down, and an alarm that should be logged according to its definition in the eventTable, the alarm is stored in the default log instead of "alarm log" (provided there are no other logs that are defined to log the alarm).

## 1.3 EVA Adaptations

As the basic EVA support is protocol independent, adaptations are needed for the actual management protocols. Such an adaptation is the manager's interface towards EVA. It may implement strategies for filtering events and alarms to different operators, or it may store more information associated with each event or alarm locally.

An EVA adaptation should be written as a gen_event handler module. It should be installed into the alarm_handler process.

An adaptation is free to use the Mnesia tables defined in EVA. The eventTable and alarmTable must be accessed within a Mnesia transaction, while the active alarm list (alarm) may be accessed dirty. Some functions in the module eva could be useful as well.

When an event or alarm is registered or sent, an gen_event event is generated as gen_event:notify(alarm_handler, ...) by EVA, and should be taken care of in the handle_event function of the adaptation.

### 1.3.1 Example

Without going too deep into the details, the following is an example of a very simple adaptation, that prints all events and alarms to standard out, and provides a function to print the active alarm list.

```
-module(simple_adaptation).

-behaviour(gen_event).

-include_lib("eva/include/eva.hrl").
-include_lib("mnemosyne/include/mnemosyne.hrl").

%%%----------------------------------------------------------------------
%%% Simple EVA adaptation that formats events and alarms to standard
%%% out.
%%%----------------------------------------------------------------------

%% External exports
-export([start/0, print_alarms/0]).

%% Internal exports
-export([init/1, handle_event/2, handle_call/2, handle_info/2, terminate/2]).

%%%----------------------------------------------------------------------
%%% API
%%%----------------------------------------------------------------------
start() ->
    gen_event:add_handler(alarm_handler, ?MODULE, []).

print_alarms() ->
    gen_event:call(alarm_handler, ?MODULE, print_alarms).
```

```
%%%----------------------------------------------------------------------
%%% Call-back functions from gen_event
%%%----------------------------------------------------------------------
init(_) ->
    io:format("Initializing simple EVA adaptation...~n"),
    {ok, []}.

handle_event({send_event, #event{name = Name}}, S) ->
    X = Name, % due to bug in mnemosyne...
    Handle = query [E.generated || E <- table(eventTable),
                                   E.name = Name] end,
    {atomic, [Generated]} =
       mnesia:transaction(fun() -> mnemosyne:eval(Handle) end),

    io:format("** Event: ~w, ~w generated~n", [Name, Generated]),
    {ok, S};

handle_event({send_alarm, #alarm{name = Name, severity = Severity}}, S) ->
    X = Name, % due to bug in mnemosyne...
    Handle = query [E.generated || E <- table(eventTable),
                                   E.name = Name] end,
    {atomic, [Generated]} =
       mnesia:transaction(fun() -> mnemosyne:eval(Handle) end),

    Handle2 = query [A.class || A <- table(alarmTable),
                                A.name = Name] end,
    {atomic, [Class]} =
       mnesia:transaction(fun() -> mnemosyne:eval(Handle2) end),

    io:format("** ~w alarm ~w of class ~w, ~w generated~n",
              [Severity, Name, Class, Generated]),
    {ok, S};

handle_event(_, S) ->
    {ok, S}.

handle_call(print_alarms, S) ->
    Handle = query [{A.name, A.sender, A.cause, A.severity, AlarmDef.class} ||
                        A <- table(alarm),
                        AlarmDef <- table(alarmTable),
                        A.name = AlarmDef.name] end,
    {atomic, Alarms} = mnesia:transaction(fun() -> mnemosyne:eval(Handle) end),
    io:format("** Active alarm list~n"),
    lists:foreach(
      fun({Name, Sender, Cause, Severity, Class}) ->
              io:format("~14w ~10w alarm ~p from ~p, probable cause: ~p~n",
                        [Severity, Class, Name, Sender, Cause])
      end, Alarms),
    {ok, ok, S}.

handle_info(_, S) ->
    {ok, S}.
```

```
terminate(R, S) ->
    io:format("Terminating simple EVA adaptation...~n"),
    ok.
```

## 1.4 EVA SNMPIinterface

This chapter describes an EVA adaptation for SNMP and an SNMP interface towards the Log Control service. Also included are an SNMP interface towards the logging of event, and an SNMP interface towards the snmp audit trail log.

An application that uses the logs or event generating function does not have to know that the events or alarms are sent as SNMP traps, it just uses the EVA API.

There are four MIBs defined, `OTP-EVA-MIB`, `OTP-LOG-MIB`, `OTP-EVA-LOG-MIB` and `OTP-SNMPEA-LOG-MIB`. These MIBs can be found in the `mibs` directory in the EVA distribution. They are described in the following sections.

### 1.4.1 EVA SNMP Adaptation

The EVA SNMP adaptation consists of functionality for translating the EVA events and alarms to SNMP traps, an SNMP MIB for the EVA tables, such as the active alarm list, and an API to be used for the SNMP instrumentation functions.

#### OTP-EVA-MIB

This MIB implements managed objects for the basic EVA service in OTP. It consists of the Event, Alarm and CurrentAlarms groups.

**Event Group** The Event group consists of the `eventTable`.

The `eventTable` has one entry for each event the system may generate. It defines all events in the system and controls how an event should be treated, and to whom it should be sent. Note that an alarm is a special kind of event, so all alarms are defined in this table as well.

The table has the following attributes:

- `eventIndex`
- `eventTrapName`
- `eventTreatment`
- `eventCommunity`
- `eventSentTraps`
- `eventOwner`

Each event has a unique index, eventIndex, which remains constant as long as the system is up and running. If some events are deleted (e.g. due to a new software release), the row will disappear.

The eventTrapName attribute defines which SNMP trap is associated with an event. This is for the manager to correlate incoming traps with the events.

The eventTreatment defines if the event should be sent as a trap or not, and if the event should be logged or not. The possible values are none, log, snmpTrap, logAndTrap. This attribute is writable. This makes it possible for the manager to select which events should be reported as traps at a specific time, and to effectively make sure that an event will not be logged in any log. How the events are logged and how to control the logs is not defined in this MIB. One log mechanism is defined in the OTP-EVA-LOG-MIB, but others could be defined instead.

The eventCommunity defines to which managers the trap should be sent, if at all. This attribute is writable.

The eventSentTraps counts the number of times the event has been sent as an SNMP trap. A manager may poll this value to see if he has lost an event.

Finally, the eventOwner is the manager entity that 'owns' the event, and is therefore responsible for its configuration. This attribute is writable.

### Alarm Group

The Alarm group consists of the alarmTable.

alarmTable is an extension to the eventTable. It has one entry for each defined alarm in the system.

The table is indexed by eventIndex and has the following attributes:

- alarmClass
- alarmSeverity

The alarmClass and alarmSeverity have the values defined by EVA. alarmSeverity is writable.

### CurrentAlarm Group

The CurrentAlarm group consists of two scalar variables, numberOfCurrentAlarms, and currentAlarmLastTimeChanged, the table currentAlarmTable and the event (defined as a trap) alarmCleared.

The numberOfCurrentAlarms is the number of active alarms in the currentAlarmTable.

The currentAlarmLastTimeChanged is a time stamp when the currentAlarmTable was changed. The time the table is changed is sent in each trap. A manager may store this value internally, and poll the currentAlarmTable variable regularly. If the internally stored value differs from the value of this variable, some alarm was lost (or not sent to the manager). In this case, the manager may download the entire table.

The currentAlarmTable is a list of all currently active alarms in the system. All objects in the table except for alarmSeverity, are read-only.

The table has the following attributes:

- currentAlarmFaultId
- currentAlarmEventIndex
- currentAlarmObject
- currentAlarmCause
- currentAlarmSeverity
- currentAlarmTime
- currentAlarmInformation

- currentAlarmExtra1
- currentAlarmExtra2

Each active alarm has a unique index, currentAlarmFaultId, which remains constant as long as the system is up and running. When an alarm is cleared, the fault id may be reused by another alarm, but only if the new alarm originates from the same fault. If the system reboots, the currentAlarmTable is reset, and all alarms that are still active are sent as new alarms.

currentAlarmEventIndex is a pointer into the eventTable. It connects the alarm to a certain trap.

currentAlarmObject defines which object generated the alarm. It should point to an instance of an accessible object in the MIB. For example, if the alarm was generated by interface no 3 in ifTable, the object should be {ifIndex 3}.

currentAlarmCause describes the cause of the alarm, if known. This is an OBJECT IDENTIFIER, which means that the possible causes must be defined in the MIB. If unknown, this object is {0 0}.

currentAlarmSeverity is the perceived severity of the alarm. The only value that can be written into this object is clear. When set to clear, the alarm is cleared and removed from the active alarm list. A clear_alarm event is generated by EVA. A management application should use this with care. Normally the application that generated an alarm is responsible for clearing the alarm.

currentAlarmTime is the time the alarm was generated. This value is written into currentAlarmLastTimeChanged when the alarm is sent.

currentAlarmInformation is a string with extra information pin-pointing the problem. Use this string with care, as too complicated strings makes it hard for a management application to make automated decisions. The only option may be to display the string to the operator as is.

currentAlarmExtra1 and currentAlarmExtra2 are extra parameters used by alarms at their own discretion. Can be used for example to identify additional objects in the alarm, or instead of currentAlarmInformation to pin-point the problem, if the additional information is defined in some MIB.

When an alarm is cleared, either by the application itself, or by an operator, the event alarmCleared is sent. In this event, one single variable is sent, currentAlarmEventIndex. Note that the currentAlarmFaultId is implicit in the instance OBJECT IDENTIFIER for this variable.

### API

The applications generate events and alarms using the API provided by EVA. They are not aware that the events and alarms are sent as SNMP traps to SNMP managers.

However, each trap that should be sent must be defined in an SNMP MIB, and there must be instrumentation functions that translates the EVA events and alarms into SNMP traps. Normally, each event and alarm in the system is mapped to separate SNMP traps. This mapping is done when the events are registered. The following functions are available for the registration. They could be called e.g. when the corresponding MIB is loaded. They are described in detail in the Reference Manual, eva, the module eva_snmp_adaptation [page 52].

`register_events([{Name, Trap, EFunc, Treatment, Community}]` This function is used to associate each event with the corresponding trap and an Erlang instrumentation function that translates the #event into a trap. It also defines the default treatment and community.

`register_alarms([{Name, Trap, AFunc, Treatment, Community}]` This function is used to associate each alarm with the corresponding trap and an Erlang instrumentation function that translates the #alarm into a trap. It also defines the default treatment and community.

In these functions, the instrumentation functions should be defined as:

```
EFunc = fun(#event) -> {ok, SnmpVarbinds}
AFunc = fun(#alarm) -> {ok, ObjOID, CauseOID, SnmpVarbinds}
```

respectively, where `SnmpVarbinds` is a list of any extra SNMP variables included in the trap.

### MIB Definition Rules

When using this SNMP EVA adaptation, each event and alarm must be defined as an SNMP trap in a MIB. Any SNMP trap may be used as an event since there are no restrictions on these traps. However, for each alarm, there are some objects that *must* be present in the trap definition. These objects must be the first objects in the trap, and they must be defined in the following order:

- `currentAlarmTime`
- `currentAlarmSeverity`
- `currentAlarmObject`

Note that implicit in each of these objects is the `currentAlarmFaultId`, since this is the index for the table.

These are the objects that are most important for the manager. Any other object may be retrieved by sending a GET request to the agent.

An example of a correct trap definition using SNMP v1 syntax:

```
boardFailure TRAP-TYPE
    ENTERPRISE board
    VARIABLES {
        currentAlarmTime,
        currentAlarmSeverity,
        currentAlarmObject,
        boardName
    }
    DESCRIPTION
        "An alarm sent when a board failure is detected."
    ::= 3
```

And the same trap using SNMP v2 syntax:

```
boardFailure NOTIFICATION-TYPE
    OBJECTS {
        currentAlarmTime,
        currentAlarmSeverity,
        currentAlarmObject
        }
    STATUS  current
    DESCRIPTION
        "An alarm sent when a board failure is detected."
    ::= { board 0 3 }
```

The values of these mandatory objects are set by EVA.

Note that after the three mandatory objects, any other objects may be specified.

### Example

This section shows an example of how EVA may be used by an application. The complete code is available in the `example` directory in the distribution.

The example application is an application that controls boards and generates an event when a board is removed or inserted, and an alarm if a board failure is detected.

The following code is the SNMP independent resource code:

```
%%%----------------------------------------------------------------
%%% Resource code
%%%----------------------------------------------------------------
reg() ->
    eva:register_event(boardRemoved, true),
    eva:register_event(boardInserted, false),
    eva:register_alarm(boardFailure, true, equipment, minor).

remove_board(No) ->
    eva:send_event(boardRemoved, {board, No}, []).

insert_board(No, BoardName, BoardType) ->
    eva:send_event(boardInserted, {board, No}, {BoardName, BoardType}).

board_on_fire(No) ->
    FaultId = eva:get_fault_id(),
    %% Cause = fire, ExtraParams = []
    eva:send_alarm(boardFailure, FaultId, {board, No}, fire, []),
    FaultId.
```

The function `reg/0` is used to register the events and the alarm in EVA. The `boardRemoved` event just identifies the removed board, but the `boardInserted` identifies the board and sends the name and type of the board as extra parameters in the event.

When this is mapped to SNMP, the following MIB is designed:

```
BOARD-MIB DEFINITIONS ::= BEGIN

IMPORTS
        DisplayString
                FROM RFC1213-MIB
        OBJECT-TYPE
                FROM RFC-1212
        experimental
                FROM RFC1155-SMI
        currentAlarmTime, currentAlarmSeverity, currentAlarmObject
                FROM OTP-EVA-MIB;

board   OBJECT IDENTIFIER ::= {experimental 1}

boardTable OBJECT-TYPE
        SYNTAX SEQUENCE OF BoardEntry
        ACCESS not-accessible
        STATUS mandatory
        DESCRIPTION
```

```
                "Contains information about the boards in the system."
        ::= { board 1 }

boardEntry OBJECT-TYPE
        SYNTAX BoardEntry
        ACCESS not-accessible
        STATUS mandatory
        DESCRIPTION
                "A set of parameters for boards."
        INDEX { boardIndex }
        ::= { boardTable 1 }

BoardEntry ::= SEQUENCE {
        boardIndex              INTEGER,
        boardName               DisplayString,
        boardType               DisplayString
        }

boardIndex OBJECT-TYPE
        SYNTAX INTEGER
        ACCESS not-accessible
        STATUS mandatory
        DESCRIPTION
                "A unique index identifying each board."
        ::= { boardEntry 1 }

boardName OBJECT-TYPE
        SYNTAX DisplayString
        ACCESS read-only
        STATUS mandatory
        DESCRIPTION
                "The name of the board."
        ::= { boardEntry 2 }

boardType OBJECT-TYPE
        SYNTAX DisplayString
        ACCESS read-only
        STATUS mandatory
        DESCRIPTION
                "The type of the board."
        ::= { boardEntry 3 }

-- Events

boardRemoved TRAP-TYPE
        ENTERPRISE board
        VARIABLES {
                boardName
                }
        DESCRIPTION
                "An event sent when a board is removed."
        ::= 1
```

```
boardInserted TRAP-TYPE
        ENTERPRISE board
        VARIABLES {
                boardName,
                boardType
                }
        DESCRIPTION
                "An event sent when a board is inserted."
        ::= 2

-- Alarms

boardFailure TRAP-TYPE
        ENTERPRISE board
        VARIABLES {
                currentAlarmTime,
                currentAlarmSeverity,
                currentAlarmObject,
                boardName
                }
        DESCRIPTION
                "An alarm sent when a board failure is detected."
        ::= 3

-- Causes

fire OBJECT IDENTIFIER ::= {board 2}
-- DESCRIPTION
--      "The board is on fire."

END
```

To implement this MIB, instrumentation functions for the managed objects are needed for the SNMP agent. Also, we must write instrumentation functions for the traps for EVA.

```
%%%----------------------------------------------------------------------
%%% SNMP adaptation code
%%%----------------------------------------------------------------------
mgm_init() ->
    snmp:load_mibs(snmp_master_agent, ["BOARD-MIB"]),
    Events = [{boardRemoved, boardRemoved, snmpTrap, "standard trap",
               {?MODULE, boardRemoved}},
              {boardInserted, boardInserted, snmpTrap, "standard trap",
               {?MODULE, boardInserted}}],
    Alarms = [{boardFailure, boardFailure, snmpTrap, "standard trap",
               {?MODULE, boardFailure}}],
    eva_snmp_adaptation:register_events(Events),
    eva_snmp_adaptation:register_alarms(Alarms).


%%----------------------------------------------------------------------
%% instrumentation functions
%%----------------------------------------------------------------------
```

```
% Using default instrumentation

%%----------------------------------------------------------------------
%% "backwards" instrumentation functions  event -> trap
%%----------------------------------------------------------------------
boardRemoved(#event{sender = {board, Idx}}) ->
    [#boardTable{name = Name}] = mnesia:dirty_read({boardTable, Idx}),
    {ok, [{boardName, [Idx], Name}]}.

boardInserted(#event{sender = {board, Idx}, extra = {Name, Type}}) ->
    {ok, [{boardName, [Idx], Name},
          {boardType, [Idx], Type}]}.

boardFailure(#alarm{sender = {board, Idx}, cause = Cause}) ->
    [#boardTable{name = Name}] = mnesia:dirty_read({boardTable, Idx}),
    {value, Oid} = snmp:name_to_oid(boardName),
    {value, COid} = snmp_cause(Cause),
    {ok, {Oid ++ [Idx], COid, [{boardName, [Idx], Name}]}}.

snmp_cause(fire) -> snmp:name_to_oid(fire);
snmp_cause(_) -> [0,0].
```

### 1.4.2 LOG SNMP Interface

The LOG SNMP interface consists of functionality for controlling the logs in the system using SNMP, and an SNMP MIB which also includes functions for tranferring logs to a remote host with FTP.

#### OTP-LOG-MIB

This MIB implements managed objects for the Log Control service. It consists of the logGroup, logTransferGroup, and the logAlarmsGroup.

**Log Group** The Log group consists of the table `logTable`.

The `logTable` has one entry for each log in the system.

Applications can choose to extend this table, for logs of certain types. This can be used e.g. to specify additional parameters for what should be logged in a log. The `evaLogDiscriminatorTable` is such an example.

The `logTable` has the following attributes:

- `logIndex`
- `logName`
- `logType`
- `logAdminStatus`
- `logOperStatus`
- `logMaxSize`
- `logNumberOfRecords`
- `logMinWrapTime`

- logWrapPercentage
- logOwner
- logRowStatus

Each log is identified by a unique index, logIndex.

logName is a string that gives a human readable name for the log. This attribute is writable at creation time. The name must be unique.

logType is an OBJECT IDENTIFIER that specifies what type of log it is. This attribute is writable at creation time. If it is an unknown log type, this entry has the value 0.0.

logAdminStatus can be up or down. Specifies the desired logOperStatus. This attribute is writable.

logOperStatus can be up or down. Specifies whether the log is active or not. A log that is down discards all log records sent to it.

logMaxSize defines the maximum size the log may occupy. When the max size is reached, logWrapPercentage of the log space is freed to make room for more records. This attribute is writable at creation time. If logTotalMaxSize + logMaxSize > logTotalMaxAllowedSize, the creation fails.

logNumberOfRecords counts the number of records in the log.

logMinWrapTime defines the minimum time between two wrap situations. If the log wraps more often, an logWrapAlarm is sent. This attribute is writable at creation time.

logWrapPercentage defines how many percent of the log space is freed when the log reaches its maximum size. This attribute is writable at creation time.

logOwner is the manager entity that 'owns' the log, and is therefore responsible for its contents, including entries in the logDiscriminatorTable. Logs created by the agent system have this object equal to "local", and should not be deleted or otherwise modified by a manager. This attribute is writable at creation.

logRowStatus is used to create and delete logs.

Log Alarms Group Two alarms are defined, the logWrapAlarm which is sent if a log wraps too often and logMediaErrorAlarm which is sent if the logging function detects an error in the storage media for a log, and cannot log anything more.

There are two probable causes defined for the logMediaErrorAlarm. These are logNoSpaceLeft which is used when there is no space left on the media, and logMediaBroken which is used when the storage media is broken.

Log Transfer Group The Log Transfer group consists of the table logTransferTable.

The logTransferTable has one entry for each transfer in the system. When a transfer entry has been created (with createAndWait), its status is notInService. When it is made active, the log transfer begins. When the transfer is complete, the status is notInService again. The outcome of the transfer session is available in the variable logTransferLastResult.

Applications can choose to extend this table, for logs of certain types. This can be used e.g. to specify additional log specific filtering parameters. The snmpeaLogTransferTable is such an example.

The logTransferTable has the following attributes:

- logTransferIndex
- logTransferStartTime
- logTransferStopTime
- logTransferFTPAddress

- logTransferFTPUser
- logTransferFTPPasswd
- logTransferFTPFile
- logTransferLastResult
- logTransferRowStatus

A log transfer entry refers to a particular log in the logTable. There may exist several log transfer entries for each log. Thus, the logTransferTable is indexed by logIndex and a logTransferIndex.

logTransferStartTime is a DateAndTime variable that specifies that log records generated after this time should be transferred. This attribute is writable.

logTransferStopTime is a DateAndTime variable that specifies that log records generated before this time should be transferred. This attribute is writable.

logTransferFTPAddress is the IP address of the remote host to which the log should be transferred. This attribute is writable.

logTransferFTPUser is the user in the FTP session. This attribute is writable.

logTransferFTPPasswd is the password for the user in the FTP session. This attribute is writable. If it is read, the empty string is returned.

logTransferFTPFile is a string with the absolute file name for the log at the remote host. This attribute is writable.

logTransferLastResult is an enumerated integer that contains the result of the last transfer. This attribute is read-only. The following values are valid:

ok the transfer succeded

aborted the transfer was aborted by the management station

ftpBadAddress the FTP address could not be contacted

ftpLoginError the combination of FTP user and passwd was invalid

ftpWriteError the user had no write access to the file

ftpTransferError the FTP session aborted

otherError any other error, e.g. internal error in log

logTransferRowStatus controls the creation/deletion of transfer entries, and controls the transfer of logs. If set to active, the transfer begins. If an active transfer's status is set to notInService, the transfer aborts. This attribute is writable.

### Creation of Local Logs

The system may choose to create local logs, i.e. logs that cannot be modified by a manager. For example, an alarm log can be created that always logs all alarms. To create a local log, the functions in log should be used. All logs that log knows of will be visible in the logTable.

### Manager Use Cases

This section describes how a manager may use the OTP-LOG-MIB to perform logging.

**Log Creation** When a management application wants to create a log, it should perform the following steps.

1. Decide which type of log is wanted. This is defined in MIBs for applications that uses the generic log control service.
2. Find a free `logIndex` by looping through the `logTable`.
3. Choose a name for the log, choose the maximum size, wrap percentage and minimum wrap time for the log.
4. Send a SET request with these parameters and `logRowStatus = createAndGo` to the agent.
5. If the creation failed because the `logIndex` was occupied, choose a new `logIndex`.

**Log Deletion** When a management application wants to delete a log, it should perform the following steps.

1. Set the `logRowStatus` to `destroy` for the corresponding `logIndex`.

**Controlling Logs** Sometimes it can be useful to block a specific log so that no records are stored in the log, but not delete it. To accomplish this a manager should:

1. Set the `logAdminStatus` to `down` for the corresponding `logIndex`.

**Log Transfer Creation** When a log should be transferred to a remote host, the following steps should be followed.

First, create the log transfer entry:

1. Decide the `logIndex` of the log that should be transferred.
2. Choose a free `logTransferIndex` by looping through all transfer entries with the same log index as the selected log.
3. Decide general filtering parameters for the selected log. This means find values for `logTransferStartTime` and `logTransferStopTime`.
4. Choose a remote host, user, password and filename.
5. Send a SET request with these parameters and `logTransferRowStatus = createAndWait` to the agent.
6. Decide log specific filtering parameters for the selected log. This depends on if the type of the log has defined additional filtering parameters.

When the log transfer entry is created, the log will be transferred each time the row is activated:

1. Send a SET request with `logTransferRowStatus = active` to the agent.
2. Poll the value of `logTransferRowStatus` until it becomes `notInService`.
3. Check the value of `logTransferLastResult` if the transfer succeeded or not.

If a log transfer takes too long time, the transfer may be aborted in the following way:

1. Send a SET request with `logTransferRowStatus = notInService` to the agent.
2. Poll the value of `logTransferRowStatus` until it becomes `notInService`. The `logTransferLastResult` is now `aborted`.

#### API

The applications create logs using the API provided by the Log Control service. However, if an application has defined additional managed objects in a MIB, the SNMP adaptation of the generic log service must know of this, in order to use this information when logs are created or trasnferred.

There is just one function needed, and it is `log_snmp:register_type/3`. It registers the type of log in the SNMP log adaptation. The function is described in detail in the reference manual.

### 1.4.3 EVA-LOG SNMP Interface

The EVA-LOG SNMP interface consists of functionality for controlling the logging of events and alarms using SNMP, and an SNMP MIB. This functionality uses the generic log control service described above.

#### OTP-EVA-LOG-MIB

This MIB implements managed objects for the EVA LOG service. It consists of the Eva Log group.

**Eva Log Group** The Eva Log group consists of the table `evaLogDiscriminatorTable`, and the variables `evaLogTotalMaxSize` and `evaLogTotalMaxAllowedSize`.

The `evaLogDiscriminatorTable` has the following attributes:

- `evaLogDiscrEventIndex`
- `evaLogDiscrRowStatus`

Each entry in this table is indexed by `logIndex` and `evaLogDiscrEventIndex`. Each row means that the event with `eventIndex` equal to `evaLogDiscrEventIndex` should be logged in the log with `logIndex`. The `evaLogDiscrRowStatus` is used to create and delete rows in the table.

The variable `evaLogTotalMaxSize` is the sum of specified maximum sizes of all logs. This object is read-only.

The variable `evaLogTotalMaxAllowedSize` is the total size all event logs created by the manager are allowed to fill. This object corresponds to the amount of disk space available for the log function in the agent system. This object is read-only.

#### Manager Use Cases

This section describes how a manager may use the OTP-EVA-LOG-MIB to perform event and alarm logging.

**Log Creation** When a management application wants to create an event log, it should perform the following steps.

1. Create a log in the `logTable` as described above, using the `logType evaLogType`.
2. If the creation succeed, decide which events should be logged in the log.
3. For each such event, create the corresponding row in `evaLogDiscriminatorTable`, using the same `logIndex` as defined in step 1

**Log Deletion** When a management application wants to delete an event log, it should follow the steps defined above in the section about general log deletion.

**Controlling Logs** Sometimes it can be useful to block a specific log so that no events are stored in the log, but not delete it. To accomplish this a manager should:

1. Set the `logAdminStatus` to `down` for the corresponding `logIndex`.

In other situations it can be useful to make a certain event not be stored in any log at all, for example if the event is generated very often. This can be accomplished by:

1. Set the `eventTreatment` to `snmpTrap` or `none` for the event in `eventTable`. As long as the `eventTreatment` is not `log` or `logAndTrap`, the event is not stored in any log.

### 1.4.4 SNMPEA LOG SNMP Interface

The SNMPEA LOG SNMP interface consists of functionality for controlling the audit trail logging mechanisms in the Extensible Snmp Agent in the system. This functionality uses the generic log control service described above.

#### OTP-SNMPEA-LOG-MIB

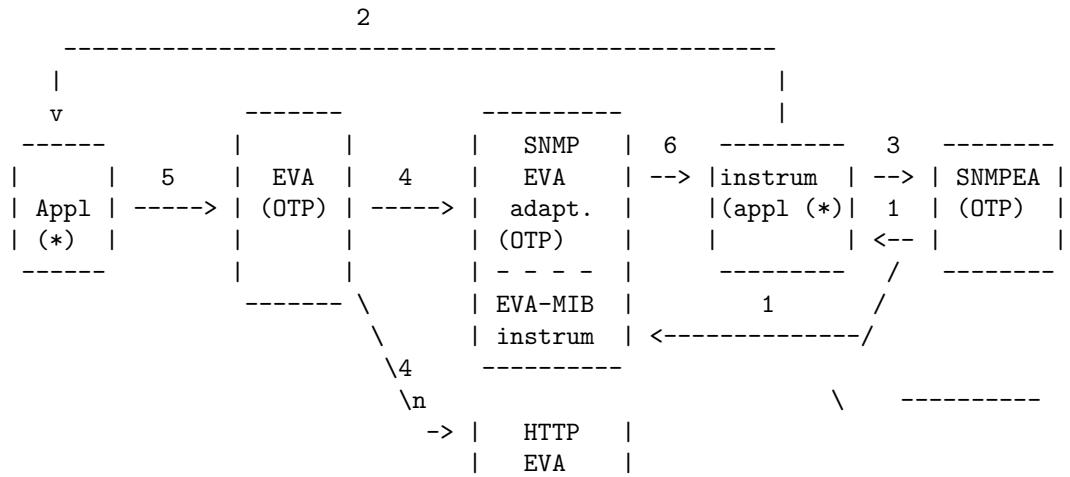
This MIB implements managed objects for the SNMEPA LOG service. It consists of the Snmpea Log group and Snmpea Log Transfer group.

**Snmpea Log Group** The Snmpea Log group consists of the single variable `snmpeaLogDiscriminator`. This variable controls which `snmpeaLogTransferTable` that extends the `logTransferTable`. It consists of a single column, `snmpeaLogTransferIPAdress`, which is used as a log specific filtering parameter. If this variable is set for a transfer entry when the log is trasnferred, requests to or from this address only are transferred.

## 1.5 Appendix A

This section describes the interfaces involved in the operations and maintenance functions for EVA. As an example SNMP is used for managament interaction.

### 1.5.1 Interfaces




The (*) marked blocks are in the application domain; the left-most is the actual application implementation, are the right-most is the instrumentation of the application.

There are several interfaces involved:

1 From SNMPEA to instrumentation functions for the application. This interface is defined in SNMPEA. [e.g. table_func(get_next, RowIndex, Cols)]

2 From instrumentation code to the resources. This interface is internal to the application, and may differ between applications.

3 From instrumentation code for events and alarms to SNMPEA traps. This interface is defined in SNMPEA. [e.g. snmp:send_trap(Trap)]

4 From EVA to the different adaptations. This interface is defined in EVA. [e.g. gen_event:notify(#alarm{})]

5 From applications to EVA. This interface is defined in EVA. [e.g. eva:send_alarm(Name, Sender, ...)]

6 From SNMP EVA to application instrumentation. This interface is defined in EVA. [e.g. board_failure(#alarm{})]

## 1.6 EVA Release Notes

This document describes the changes made to the EVA application.

### 1.6.1 EVA - Event and Alarm Handler v2.0.2.1

#### Improvements and New Features

- EVA is now able to handle upgrade properly.

#### Fixed Bugs and Malfunctions

-

#### Incompatibilities With Event and Alarm handler v2.0.2

-

#### Known Bugs and Problems

-

### 1.6.2 EVA - Event and Alarm Handler v2.0.2

#### Improvements and New Features

- A duplicated entry in a case-statement, i.e., a few lines of unused code, has been removed.

#### Fixed Bugs and Malfunctions

-

#### Incompatibilities With Event and Alarm handler v2.0.1

-

#### Known Bugs and Problems

-

### 1.6.3 EVA 2.0.1

#### Reported Fixed Bugs and Malfunctions

- The configuration parameter use_snmpea_log didn't work.
  Own Id: OTP-2117
- The SNMP trap `alarmCleared` in OTP-EVA-MIB was erroneously defined and sent. The `currentAlarmEventIndex` is sent in the trap now.
  Own Id: OTP-2118
- The MODULE-IDENTITY was erroneously named in OTP-EVA-MIB. It is now corrected to `otpEvaModule`.
  Own Id: OTP-2136

### 1.6.4 EVA 2.0.0

#### Improvements and new features

- The log functionality is made generic. Any log in the system can be controlled by the log service in eva.
- Any log can be filtered and formatted and transferred with FTP to a remote host.
- A timestamp is added to each event.
- The fault id for an alarm does not have to be generated by `eva:get_fault_id/0`.
- The default log argument in `eva_log:start_link/1`, `eva_sup:start_link_log/1`, `eva_sup:start_link_log_snmp/3` and `eva_log_sup:start_link/1,3` is now a 2-tuple {`Name`, `WrapTime`}. If the, old style is used, `WrapTime` defaults to 86400 seconds (i.e. 24 h)

#### Reported Fixed Bugs and Malfunctions

- The source code in the examples directory now compile!
  Own Id: OTP-1766
- Errors in logging was not handled correctly; the operational status was not updated.
  Own Id: OTP-1769

### Incompatibilities with 1.0

- The generic log functionality is moved from the module `eva_log` to the module `log`.
- The generic SNMP log functionality is moved from `OTP-EVA-LOG-MIB` to `OTP-LOG-MIB`.

## 1.6.5 EVA 1.0

New application.

# EVA Reference Manual

## Short Summaries

- Application eva [page 30] – The Event and Alarm Handling Application
- Erlang Module eva [page 32] – Client API for the Event and Alarm handling Functionality in the EVA Application
- Erlang Module eva_log [page 38] – Log functionality for events and alarms in EVA
- Erlang Module eva_server [page 40] – The Main Global Server in EVA
- Erlang Module eva_sup [page 41] – A Supervisor for the EVA Application
- Erlang Module log [page 43] – Client API for the log functionality in the EVA application
- Erlang Module log_server [page 47] – The main server in LOG

### eva

No functions are exported.

### eva

The following functions are exported:

- `aclear_alarm(FaultId)`
  [page 34] Clear an alarm
- `clear_alarm(FaultId)`
  [page 34] Clear an alarm
- `clear_alarm(FaultId, Time) -> ok`
  [page 34] Clear an alarm
- `get_alarm_status() -> [{Severity, boolean()}]`
  [page 34] Return information on whether there is any active alarm or not
- `get_alarms(Item) -> [#alarm]`
  [page 34] Return all active alarms matching `Item`
- `get_fault_id() -> fault_id()`
  [page 34] Return a fault id for a new alarm
- `get_no_alarms() -> integer()`
  [page 35] Return the number of active alarms in the system
- `register_alarm(Name, Log, Class, Severity) -> boolean()`
  [page 35] Register an alarm within EVA

- `register_event(Name, Log) -> boolean()`
  [page 35] Register an event within EVA
- `asend_alarm(Name, FaultId, Sender, Cause, Extra)`
  [page 35] Send an alarm and makes it active
- `send_alarm(Name, FaultId, Sender, Cause, Extra)`
  [page 35] Send an alarm and makes it active
- `send_alarm(Name, FaultId, Sender, Cause, Extra, Time) -> ok | {error, Reason}`
  [page 35] Send an alarm and makes it active
- `asend_event(Name, Sender, Extra)`
  [page 36] Send an event
- `send_event(Name, Sender, Extra)`
  [page 36] Send an event
- `send_event(Name, Sender, Extra, Time) -> ok | {error, Reason}`
  [page 36] Send an event
- `unregister_alarm(Name) -> void()`
  [page 36] Unregister an alarm within EVA
- `unregister_event(Name) -> void()`
  [page 37] Unregister an event within EVA
- `alarm_first() -> {ok, Index} | '$end_of_table'`
  [page 37] Return the index of the first element in the alarm table
- `alarm_next(Index) -> {ok, NextIndex} | '$end_of_table'`
  [page 37] Return the next index in the alarm table

## eva_log

The following functions are exported:

- `close(Name) -> ok`
  [page 38] Close an event log
- `get_logs() -> [#eva_log]`
  [page 38] Return all event logs
- `open(Name, FilterFunction, WrapTime) -> ok | {error, Reason}`
  [page 38] Open a new event log
- `set_filter(Name, FilterFunction)`
  [page 39] Change the filter function for a log
- `start_link()`
  [page 39] Start the eva log service
- `start_link(DefaultLog) -> {ok, Pid} | {error, Reason}`
  [page 39] Start the eva log service

## eva_server

The following functions are exported:

- `create_tables(Nodes) -> void()`
  [page 40] Creates the Mnesia tables required for the eva server
- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 40] Start the eva server

### eva_sup

The following functions are exported:

- `create_tables(Nodes) -> void()`
  [page 41] Create Mnesia tables for basic EVA
- `create_tables_log(Nodes) -> void()`
  [page 41] Create Mnesia tables for basic EVA and log
- `create_tables_log_snmp(Nodes) -> void()`
  [page 41] Create Mnesia tables for basic EVA, log, and SNMP implementations
- `create_tables_snmp(Nodes) -> void()`
  [page 41] Create Mnesia tables for basic EVA and SNMP implementation
- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 42] Start basic EVA
- `start_link_log(DefaultLog) -> {ok, Pid} | {error, Reason}`
  [page 42] Start basic EVA and log
- `start_link_log_snmp(DefaultLog, LogDir, MaxDirSize) -> {ok, Pid} | {error, Reason}`
  [page 42] Start basic EVA, log and SNMP implementations
- `start_link_snmp() -> {ok, Pid} | {error, Reason}`
  [page 42] Start basic EVA and SNMP implementation

### log

The following functions are exported:

- `close(Name) -> ok`
  [page 44] Close an open log
- `get_logs() -> [Log]`
  [page 44] Get all logs known to the log server
- `open(Name,Type,WrapTime)`
  [page 44] Open a log
- `set_admin_status(Name, AdminStatus) -> OperStatus | {error, Reason}`
  [page 44] Set the administrative status of the log
- `transfer(Host,User,Passwd,DestFile,SearchFunc) -> ok | {error, Reason}`
  [page 44] Transfer a log with FTP

### log_server

The following functions are exported:

- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 47] Start the log server

# eva

Application

The Event and Alarm handling appliction (EVA) is a Fault Management application that provides support to applications and managers for sending and controlling events and alarms, and for control and transfer of logs in the system.

EVA is modular and provides two different management protocol independent services. These two services are basic `eva`, which provides event and alarm definition and sending, and `log` which provides support for controlling logs in the system, and for transferring logs to remote systems. There is also a service called `eva_log`, which provides a specialization of the generic log mechanism, for logging of events and alarms. The basic `eva` can be used independently of `log`. EVA defines an API that can be used to make management protocol specific interfaces to EVA, for example SNMP, CORBA, or HTTP interfaces. Currently, an SNMP interface to the two generic services are defined.

EVA is designed to be used as an included application, which means that it needs another application to include it, in order to run. That application is an ordinary application which starts the EVA services it needs in its supervision tree. The services can either be started individually, or by using the supervisor `eva_sup`.

EVA is designed to be a distributed global application, which means that the super application that includes EVA may be specified as a distributed application which runs at one node at a time only, with the other nodes as standby nodes. The basic EVA service - the eva server - is a global server, which means that clients can access the EVA functionality from any node.

EVA uses the Mnesia DBMS to store data. This means that Mnesia must be running on all nodes where EVA can run, and that the tables EVA uses are created and configured correctly. Each EVA service provides a function that should be called to create the tables, and to define the replicas for Mnesia. Each such function is called `create_tables*` and takes one parameter that is 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 where the super application that includes EVA is defined to run as a distributed application.

## Configuration

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

`use_snmpea_log = true | false <optional>` Specifies if the audit trail log in snmp should be visible and controllable in the log application. Default is `false`.

## SNMP MIBs

The following MIBs are defined in EVA:

OTP-EVA-MIB (eva) This MIB contains objects for instrumentation and control of the events and alarms in the system.

OTP-LOG-MIB (eva) This MIB contains objects for instrumentation and control of the logs and FTP transfer of logs.

OTP-EVA-LOG-MIB (eva) This MIB contains objects for instrumentation and control of the events and alarm logs in the system.

OTP-SNMPEA-LOG-MIB (eva) This MIB contains objects for instrumentation and control of the snmp audit trail log in the system.

The MIBs are stored in the `mibs` directory. All MIBs are defined in SNMPv2 SMI syntax. SNMPv1 versions of the mibs are delivered in the `mibs/v1` directory.

The compiled MIBs are located under `priv/mibs`, and the generated `.hrl` files under the `include` directory. To compile a MIB that IMPORTS an EVA MIB, give the option `{il, ["eva/priv/mibs"]}` to the MIB compiler.

The MIBs are loaded into the agent when the services are started.

## See Also

eva(3), eva_log(3), eva_server(3), eva_sup(3), eva_log_snmp(3), eva_snmp_adaptation(3), log(3), log_snmp(3), log_snmpea(3), snmp(6)

# eva

Erlang Module

This module contains functions for the client API to the Event and Alarm handling application EVA. EVA is a distributed global application, which means that clients can access the EVA functionality from any node. There is a globally registered server called `eva_server` to which all requests are sent. The client functions for sending and clearing events and alarms exist in two variants; one asynchronous and one synchronous. The decision to use one or the other depends on how secure the delivery of events should be. If the asynchronous variant is used, the message may be lost if the node where the `eva_server` crashes after the message is sent, but before it is correctly received. The synchronous variant fails if it does not get an acknowledgment back from the server. In this case, it is up to the client application to decide what to do. It may, for example, wait a few seconds for another node to takeover the EVA application, and then try again.

An *event* is a notification sent from the *NE* to a management application. An event is uniquely identified by its name. A special form of an event is an *alarm*. An alarm represents a fault in the system that needs to be reported to the manager. An example of an alarm could be `equipment_on_fire`. When an alarm is sent, it becomes active and is stored in an *active alarm list*. When the application from which the alarm was sent notices that the fault that caused the alarm is not valid anymore, it *clears* the alarm. When an alarm is cleared, the alarm is deleted from the active alarm list, and an `clear_alarm` event is generated by EVA. Each fault may give rise to several alarms, maybe with different severities. There can, however, only be one active alarm for each fault at any one time. For example, associated with disk space usage may be two alarms, `disk_80_percent_filled` and `disk_90_percent_filled`. These two alarms represents the same fault, but only one of them can be active at the same time. An active alarm is identified by its *fault_id*. In contrast to alarms, ordinary events do not represent a fault, and they are not stored as the alarms in the active alarm list.

The basic EVA server is a global server to which all events and alarms are sent. The server updates its tables, the active alarm list for example, and sends the event or alarm to the `alarm_handler` process that runs on the same node as the global server. `alarm_handler` is a `gen_event` process defined in SASL.

EVA stores the definitions of events and alarms in the Mnesia tables `eventTable` and `alarmTable` respectively. As an alarm is a special form of an event, each alarm is present in both of these tables. The active alarm list is stored in the Mnesia table `alarm`. The records for all these tables are defined in the header file `eva.hrl`, available in the `include` directory in the distribution.

The EVA application provides functionality to send and to log events and alarms. The logs can be examined by a manager at a later time.

Before a client can send any events or alarms, the name of the event must be registered in EVA. To register an event, a client calls `register_event/2`. The parameters of this function are the name of the event and notification of whether the event should be logged by default or not. A manager can decide to change this value later. To register an

alarm, a client calls register_alarm/4. The parameters of this function are the name and logging parameters as for events, and the class and default severity of the alarm.

## Adaptations and Subscriptions

The EVA services are management protocol independent. However, to provide EVA services to a manager, a management protocol is however needed. EVA uses *adaptations* for mapping of EVA services to specific protocols. Adaptations need access to the Mnesia tables used in EVA.

The event definitions are stored in the Mnesia table eventTable, and the alarm definitions in alarmTable. They are replicated to disk and RAM on each node that may run the EVA application. The tables are defined as follows:

```
-record(eventTable, {name, log, generated}).
-record(alarmTable, {name, class, severity}).
```

Each alarm is defined in both the event and alarm table, since an alarm is a special kind of event. log is a boolean which defines whether the event should be logged or not, generated is a counter that is incremented each time this event is sent, class and severity are as defined in register_alarm/4 below.

The active alarm list is stored in the Mnesia table alarm. The alarm record is defined as:

```
-record(alarm, {index, fault_id, name, sender,
                cause, severity, time, extra}).
```

These records are defined in the file include/eva.hrl. To include this file in your code, use -include_lib("eva/include/eva.hrl")..

All these tables are part of the API, which means that they may be accessed and modified by any application, for example by an EVA adaptation. They must be accessed and modified within a transaction.

When an event or alarm is generated by an application, it is sent to the global eva server, which updates the Mnesia tables, and constructs a record that it sends to the local alarm_handler process in the SASL application. alarm_handler is a gen_event manager process, which means that eva server uses gen_event:notify to send the event or alarm record. An application which needs to subscribe to certain events, should write a gen_event handler module and install it in the alarm_handler. EVA adaptations should do this as well. The eva server sends the following gen_event notifications to alarm_handler:

{register_alarm, Name} Sent when an alarm has been registered.

{register_event, Name} Sent when an event has been registered.

{send_alarm, #alarm} Sent when an alarm is to be sent. The eva server sends this notification after the Mnesia tables have been updated. An adaptation should translate the #alarm into a format suitable for the protocol that the adaptation implements.

{send_event, #event} Sent when an event is to be sent. The eva server sends this notification after the Mnesia tables have been updated. An adaptation should translate the #event into a format suitable for the protocol the adaptation implements.

{unregister_alarm, Name} Sent when an alarm has been unregistered.

{unregister_event, Name} Sent when an event has been unregistered.

When an alarm is cleared, EVA generates an event called `clear_alarm`, where `#event.sender` is the index in the table `alarm`. For example, if an application calls `eva:clear_alarm(Fault)` and the fault was stored with index 6 in the active alarm list, the following `#event` is generated: `#event{name = clear_alarm, sender = 6}`.

The `clear_alarm` event is generated using `gen_event:sync_notify`, which means that all adaptations and subscribers are given a chance to take care of this event, before the alarm is deleted from the active alarm list.

## Exports

```
aclear_alarm(FaultId)
clear_alarm(FaultId)
clear_alarm(FaultId, Time) -> ok
```

Types:

- FaultId = fault_id()
- Time = integer() > 0 | infinity

These functions are used to clear an active alarm. The `FaultId` is a term the uniquely identifies the fault. For example, the function `get_fault_id/0` can be used to generate a unique id.

`aclear_alarm/1` is an asynchronous function which just sends the clear alarm request to the global eva server. `clear_alarm/1,2` are synchronous functions that wait `Time` ms for an answer. If `Time` is not given, it defaults to 10000 ms.

If the server does not respond within the specified time, the function exits with reason `{timeout, _}`.

```
get_alarm_status() -> [{Severity, boolean()}]
```

Types:

- Severity = severity()

For each alarm severity, it returns information on whether there is any active alarm for that severity or not.

```
get_alarms(Item) -> [#alarm]
```

Types:

- Item = {name, Name} | {sender, Sender}
- Name = atom()
- Sender = term()

Returns all active alarms which match `Item`. This function can be used by a client to check if it has any active alarms defined when it starts. For each such alarm, it must be prepared to clear it. A client may, for example, at start-up perform a “self-test” to see which alarms should be active, and compare then this with what this function returns, and clear or send missing alarms.

```
get_fault_id() -> fault_id()
```

This function can be called before a client sends an alarm to obtain a globally unique fault identity that can be used in subsequent calls to `send_alarm` and `clear_alarm`.

This function does not communicate with the `eva_server`, it just constructs a unique reference and is therefore fast.

`get_no_alarms() -> integer()`

Returns the number of active alarms in the system.

`register_alarm(Name, Log, Class, Severity) -> boolean()`

Types:

- Name = atom()
- Log = boolean()
- Class = class()
- Severity = severity()
- class() = unknown | communications | qos | processing | equipment | environmental
- severity() = indeterminate | critical | major | minor| warning

Registers an alarm within EVA. An alarm must be registered before it is sent the first time. The registration information is stored persistently, so this function can be called just once. However, if EVA detects that the alarm is already registered, it discards the registration and returns `false`. Otherwise, it returns `true`.

The `Log` parameter defines if the alarm should be logged by default or not.

The `Class` and `Severity` parameters are originally defined in X.733, ITU Alarm Reporting Function.

`register_event(Name, Log) -> boolean()`

Types:

- Name = atom()
- Log = boolean()

Registers an event within EVA. An event must be registered before it is sent the first time. The registration information is stored persistently, so this function can be called just once. However, if EVA detects that the event is already registered, it discards the registration and returns `false`. Otherwise, it returns `true`.

The `Log` parameter defines if the event should be logged by default or not.

`asend_alarm(Name, FaultId, Sender, Cause, Extra)`

`send_alarm(Name, FaultId, Sender, Cause, Extra)`

`send_alarm(Name, FaultId, Sender, Cause, Extra, Time) -> ok | {error, Reason}`

Types:

- Name = atom()
- FaultId = fault_id()
- Sender = term()
- Cause = term()
- Extra = term()
- Time = integer() > 0 | infinity

- Reason = {no_such_alarm, Name} | {aborted, Name, R}

These functions are used to send an alarm and make it active (stored in the active alarm list).

`Name` is the name of the alarm. The alarm must be registered before this function is called.

`FaultId` is a term the uniquely identifies the fault. For example, the function `get_fault_id/0` can be used to generate a unique id.

`Sender` is the object that generated the alarm. It could, for example, be a tuple {`board`, `7`} or a registered name. This object should be fairly constant - not a Pid - so that it is possible to trace the sending object at a later time.

`Cause` is the cause of the alarm. It is recommended not to use strings as cause, to make it easier to match upon for other programs. For example a management application may want to translate the cause into another language.

`Extra` is any extra information which describes the alarm.

`asend_alarm/5` is an asynchronous function which just sends the alarm request to the global eva server. `send_alarm/5,6` are synchronous functions that wait `Time` ms for an answer. If `Time` is not given, it defaults to 10000 ms.

If the server does not respond within the specified time, the function exits with reason {`timeout`, `_`}.

```
asend_event(Name, Sender, Extra)
send_event(Name, Sender, Extra)
send_event(Name, Sender, Extra, Time) -> ok | {error, Reason}
```

Types:

- Name = atom()
- Sender = term()
- Extra = term()
- Time = integer() > 0 | infinity
- Reason = {no_such_event, Name} | {aborted, Name, R}

These functions are used to send an event to the eva server.

`Name` is the name of the event. The event must be registered before this function is called.

`Sender` is the object that generated the event. It could, for example, be a tuple {`board`, `7`} or a registered name. This object should be fairly constant - not a Pid - so that it is possible to trace the sending object at a later time.

`Extra` is any extra information which describes the event.

`asend_event/3` is an asynchronous function, that just sends the event request to the global eva server. `send_event/3,4` are synchronous functions that waits `Time` ms for an answer. If `Time` is not given, it defaults to 10000 ms.

If the server does not respond within the specified time, the function exits with reason {`timeout`, `_`}.

```
unregister_alarm(Name) -> void()
```

Types:

- Name = atom()

Unregisters an alarm within EVA. This function should only be used when an alarm definition should be removed, due to a new release of the system, for example.

`unregister_event(Name) -> void()`

Types:

- Name = atom()

Unregisters an event within EVA. This function should only be used when an event definition should be removed, due to a new release of the system, for example.

## Access Functions for the Active Alarm List

The active alarm list is stored in the Mnesia table `alarm`. This table is indexed by an integer `alarmIndex`. This integer is used to get the table ordered, with the latest sent alarm after the previous. Currently ordered Mnesia tables cannot be traversed in a convenient way and for this reason this module provides two functions to handle the traversal. These functions will be removed if ordered tables are implemented in Mnesia.

## Exports

`alarm_first() -> {ok, Index} | '$end_of_table'`

Types:

- Index = integer()

Returns the index of the first element in the alarm table. This is a temporary function which will be removed if ordered tables are implemented in Mnesia.

`alarm_next(Index) -> {ok, NextIndex} | '$end_of_table'`

Types:

- Index = NextIndex = integer()

Returns the next index after `Index` in the alarm table. This is a temporary function which will be removed if ordered tables are implemented in Mnesia.

## See Also

alarm_handler(3), gen_event(3), mnesia(3)

# eva_log

Erlang Module

The EVA log service uses the generic Log Control service to implement log functionality for events and alarms defined in EVA.

In the rest of this description, the term *event* refers to both events and alarms as defined in EVA.

This log functionality supports logging of events from EVA. It uses the module `disk_log` for logging of events, using the internal log format defined by `disk_log`. There can be several logs active at the same time. Each log uses a filter function to decide whether an event should be stored in the log or not.

There are several ways to control whether an event should be stored in a log or not. First of all, `eva_log` checks if the `log` flag in `eventTable` is set (see `eva(3)`). If it is set to `false`, the event is not stored in any log, even if there are logs that are configured to log the event. In this way, logging of individual events can be turned on/off by a manager. If the `log` flag is `true`, `eva_log` checks the operational status of the log. If it is `down`, the event is not stored. If it is `up`, the associated filter function is called. If this function returns `true`, the event is stored, otherwise it is discarded. To summarize, all these conditions must be true for an event to be stored:

- The `log` flag for the event is `true`.
- The operational status for the log is `up`.
- The filter function for the log returns `true`, when applied to the event.

There is a concept of a default log. The default log is used to log any event that has the `log` flag in `eventTable` set to `true`, but no log is currently able to store the event (or there is no other log defined to log the event). The usage of the default log is optional.

## Exports

`close(Name) -> ok`

Types:

- Name = string()

Closes an event log.

`get_logs() -> [#eva_log]`

Returns all event logs known to `eva_log`. The record `#eva_log` is defined in the file `eva_log.hrl`.

`open(Name, FilterFunction, WrapTime) -> ok | {error, Reason}`

Types:

- Name = string()
- FilterFunction = {M, F, A}
- M = F = atom()
- A = list()
- WrapTime = integer()

Makes `eva_log` aware of the log `Name`. The log must be an open `disk_log` log, with log format `internal`. This function will call `log:open(Name, eva_log, WrapTime)` in order to register the log in the generic Log Control service.

The `FilterFunction` is used when an event is received from EVA. It is then called as `M:F(Event ++ A)`, and supposed to return `true` if the event should be stored in the log. All other return values makes the event be discarded. The filter function can be exchanged during runtime, by using `set_filter/2`.

`set_filter(Name, FilterFunction)`

Types:

- Name = string()
- FilterFunction = {M, F, A}
- M = F = atom()
- A = list()

Changes the filter function for the event log.

`start_link()`
`start_link(DefaultLog) -> {ok, Pid} | {error, Reason}`

Types:

- DefaultLog = {Name, WrapTime} | false
- Name = string()
- WrapTime = integer()

Starts the eva log service. This function can be used to include the service in a supervisor. Normally, functions in the supervisor `eva_sup` can be used instead.

`DefaultLog` is either the name and wrap time of the default log to use, or `false`. If it is not `false`, the log must have been created with `disk_log`, just as any other log (see `open/3`). The default log is used to log any event that has the `log` flag set to true in `eventTable`, but no log has actually logged the event, either because there was no such log, or the log had operational status `down`. If the `DefaultLog` argument is omitted, it defaults to `false`. If the default log is used, it will be made known to the generic Log Control service as `log:open(Name, eva_log, WrapTime)`.

## See Also

disk_log(3), eva(3), eva_sup(3), file(3), log(3)

# eva_server

Erlang Module

This module implements the main global server in EVA. The client API towards this server is defined in the module `eva`. The functions in this module are used to start the service.

## Exports

`create_tables(Nodes) -> void()`

Types:

- Nodes = [node()]

This function creates the Mnesia tables required for the `eva_server`. `Nodes` is a list of nodes where the tables should be replicated. This list of nodes should specify the same nodes where the application EVA can be run distributed, so that EVA always can have local access to the tables.

This function should be called once when installing the EVA application in the system.

`start_link() -> {ok, Pid} | {error, Reason}`

Starts the eva server. This function can be used to include the server in a supervisor. Normally, functions in the supervisor `eva_sup` can be used instead.

The function `create_tables/1` must have been called before the server is started.

## See Also

eva(3), eva_sup(3)

# eva_sup

Erlang Module

This module provides a supervisor for the entire EVA application. An application can use this supervisor, or write its own, using the `start_link` functions of the individual services.

The supervisor can be configured to start the different EVA services independently, such as the basic `eva_server` and `eva_log` functionality, and to start the SNMP implementations of the respective service. For each possible combination of services, there is a corresponding start function, for example `start_link_log/1` which starts the basic eva server and the log functionality.

Before the services can be used, the Mnesia tables involved must be created. For each combination of services, there is a corresponding function which creates the tables. Each such function takes a list of nodes as its argument. This list of nodes defines to which nodes the Mnesia tables will be replicated. These nodes should be the same nodes as where the application where EVA is included can run distributed. This is as per the `kernel` configuration parameter `distributed`.

## Exports

`create_tables(Nodes) -> void()`

Types:

- Nodes = [node()]

Creates the Mnesia tables for the basic EVA service (`eva_server`) only.

`create_tables_log(Nodes) -> void()`

Types:

- Nodes = [node()]

Creates the Mnesia tables for the basic EVA service (`eva_server`) and EVA log service (`eva_log`).

`create_tables_log_snmp(Nodes) -> void()`

Types:

- Nodes = [node()]

Creates the Mnesia tables for the basic EVA service (`eva_server`), EVA log service (`eva_log`), and for the SNMP implementation of these (`eva_snmp_adaptation` and `eva_log_snmp`).

`create_tables_snmp(Nodes) -> void()`

Types:

- Nodes = [node()]

Creates the Mnesia tables for the basic EVA service (`eva_server`), and for the SNMP implementation of this service (`eva_snmp_adaptation`).

`start_link() -> {ok, Pid} | {error, Reason}`

Starts the supervisor and the basic EVA service (`eva_server`) only.

`start_link_log(DefaultLog) -> {ok, Pid} | {error, Reason}`

Types:

- DefaultLog = string() | false

Starts the supervisor, the basic EVA service (`eva_server`) and EVA log service (`eva_log`).

`DefaultLog` is passed to `eva_log:start_link(DefaultLog)`.

`start_link_log_snmp(DefaultLog, LogDir, MaxDirSize) -> {ok, Pid} | {error, Reason}`

Types:

- DefaultLog = string() | false
- LogDir = string()
- MaxDirSize = integer()

Starts the supervisor, the basic EVA service (`eva_server`), EVA log service (`eva_log`) and the SNMP implementations of these (`eva_snmp_adaptation` and `eva_log_snmp`).

`DefaultLog` is passed to `eva_log:start_link(DefaultLog)`, `LogDir` and `MaxDirSize` to `eva_log_snmp:start_link(LogDir, MaxDirSize)`.

`start_link_snmp() -> {ok, Pid} | {error, Reason}`

Starts the supervisor, the basic EVA service (`eva_server`) and the SNMP implementation of this service (`eva_snmp_adaptation`).

## See Also

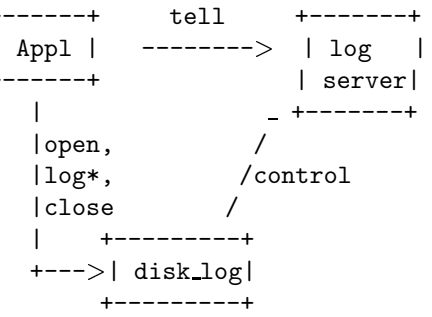
eva_log(3), eva_server(3), eva_log_snmp(3), eva_snmp_adaptation(3)

# log

Erlang Module

This module contains client functions to the generic Log Control services in the EVA application. There are two services available; log monitoring and log transfer. The logs are controlled by a log server, and each log may be transfered with FTP to a remote host.

The log server has a list of all active logs in the system. An application that wants to make a log controllable with this functionality, must register the log in the log server. Each log is implemented as a `disk_log` log. The application stores its log records using the ordinary functions in `disk_log`. The following picture illustrates the idea:




First, the application opens the log. Thenit registers the log in the log server, which makes the log server control the log. The application can store log records in the log, until it eventually closes the log, and tells the log server about it.

Each log has an administrative and an operational status, that both can be either `up` or `down`. The administrative status is configurable, and reflects the desired operational status. Normally they are both the same. If the administrative status is set to `up`, the operational status will be `up` as well. However, if the log for some reason does not work, for example if the disk partition is full, the operational status will be `down`. When the operational status is down, no records are stored in the log.

## Alarms

Two EVA alarms are defined in the `log` service, `log_file_error` and `log_wrap_too_often`.

- `log_file_error`. This alarm is generated if there is a file error when an item should be logged. Default severity is `critical`. The cause for this alarm can be any `Reason` as returned from `file:write` in case of error (it returns {`error`, `Reason`}). The alarm is cleared if the file system starts working again.
- `log_wrap_too_often`. This alarm is generated when the log wraps more often than the wrap time. Default severity is `major`. The cause for this alarm is undefined. The alarm is cleared if the log wraps within the wrap time, the next time it wraps.

## Exports

`close(Name) -> ok`

Types:

- Name = string()

Use this function to remove a log from the log server.

`get_logs() -> [Log]`

Types:

- Log = #log

Returns all logs known to `log_server`. The record `#log` is defined in the file `log.hrl`.

`open(Name,Type,WrapTime)`

Types:

- Name = string()
- Type = term()
- WrapTime = integer()

Makes `log_server` aware of the log `Name`. The log must be an open `disk_log` log.

The type argument is there for information to a manager.

If the log is a wrap log, `log_server` generates the `log_wrap_too_often` alarm if the log wraps more often than `WrapTime` seconds. In this context, wraps means that `disk_log` switches to a previously used file, and some log items are lost.

`set_admin_status(Name, AdminStatus) -> OperStatus | {error, Reason}`

Types:

- Name = string()
- AdminStatus = OperStatus = up | down
- Reason = {error, {no_such_log, Name}}

Sets the desired state of the log. Returns the new operational status of the log. If the administrative status is set to `up`, and the operational status is `down`, there is some error with the logging mechanism, for example if the disk partition is full.

If the operational status of the log is `down`, no log records will be stored in the log. This function uses the functions `disk_log:block/unblock` to change the operational status.

`transfer(Host,User,Passwd,DestFile,SearchFunc) -> ok | {error, Reason}`

Types:

- Host = ip_address()
- User = string()
- Passwd = string()
- DestFile = string()
- SearchFunc = {M,F,A}
- Reason = ftp_bad_address | ftp_login_error | ftp_write_error | ftp_tranfer_error | {bad_search_result, term()}

- ip_address() = string() | {int(), int(), int(), int()}
- M = F = atom()
- A = list()
- M:F(Continuation | A) -> SearchResult
- SearchResult = eof | {NewContinuation, Bytes} | {error, R}
- Continuation = start | cont()
- NewContinuation = cont()
- Bytes = binary()
- R = term()

This function is used to transfer a log with FTP to a remote host, for example a mangement station. It could be triggered from for example SNMP or from a web interface to the system. This log is received as one contiguous file, although it is stored as several files in the underlying `disk_log` log. It is possible to filter the log for certain log records, and to format the log records. Thus, log records can be efficiently stored by not formatting them when they are written, but later when the log is actually needed. Of course, to further improve performance, the log records can be transferred unformatted as well, and later formatted off-line at the management station.

The `Host` argument is either a string or a four-tuple representing the IP address of the host. The string can be the name of the host, or the IP address in dotted decimal notation, for example “150.236.14.136”.

The `SearchFunc` argument specifies a function that will be called by the transfer session to get a chunk of log records to transfer. At the first call, the atom `start` is used as an initial continuation. Each time the function is called, it is supposed to return a new continuation and a binary that contains the bytes to be transferred (the formatted log records). When the end of the log is reached, `eof` is returned by the function. The return values of the `SearchFunc` is chosen to match those of `disk_log:chunk/2`. The extra arguments (`A`) to the functions can be used to pass filtering information to the search function. An example of a search function:

```
-module(my_log).

f(Cont, Time) ->
    case disk_log:chunk("my_log", Cont) of
        eof ->
            eof;
        {error, R} ->
            {error, R};
        {NCont, ListOfTerms} ->
            List = lists:map(fun(Term) ->
                                format(Term, Time)
                             end, ListOfTerms),
            Bin = list_to_binary(List),
            {NCont, Bin}
    end.

%% Each log record is a tuple: {LogTime, LogData}
format({LogTime, LogData}, Time) when LogTime > Time ->
    io_lib:format("time: ~p data: ~p~n", [LogTime, LogData]);
format(_LogRecord, _Time) ->
    [].
```

This function can be used as follows to transfer all log records stored after 1997-11-01:

```
log:transfer("cave.ericsson.se", "mbj", "secret!", "my_log.txt",
             {my_log, f, [{1997,11,01}]}
```

## See Also

disk_log(3), eva(3), file(3)

# log_server

Erlang Module

This module implements the main server in the LOG application. The client API to this server is defined in the module `log`.

## Exports

`start_link() -> {ok, Pid} | {error, Reason}`

Starts the log server. This function can be used to include the server in a supervisor. Normally, functions in the supervisor `log_sup` can be used instead.

## See Also

eva_log_sup(3), log(3)

# EVA Reference Manual - SNMP adaptation

## Short Summaries

- Erlang Module **eva_log_snmp** [page 50] – Implements an SNMP Interface to EVA log
- Erlang Module **eva_snmp_adaptation** [page 52] – An SNMP Adaptation to EVA
- Erlang Module **log_snmp** [page 55] – Implements an SNMP interface to the log service in the EVA application
- Erlang Module **log_snmpea** [page 57] – SNMP instrumentation functions for the OTP-SNMPEA-LOG-MIB

## eva_log_snmp

The following functions are exported:

- `create_tables(Nodes) -> void()`
  [page 50] Create the Mnesia tables required for EVA log SNMP implementation
- `start_link(LogDir, MaxSize) -> {ok, Pid} | {error, Reason}`
  [page 50] Start the EVA LOG SNMP service
- `log_discr_table(Op, RowIndex, Cols) -> InstrumRet`
  [page 51] Instrumentation function for the logDiscriminatorTable
- `log_table(Op, RowIndex, Cols) -> InstrumRet`
  [page 51] Instrumentation function for the logTable
- `log_total_max_allowed(get) -> InstrumRet`
  [page 51] Instrumentation function for the logTotalMaxAllowedSize
- `log_total_max_size(get) -> InstrumRet`
  [page 51] Instrumentation function for the logTotalMaxSize

## eva_snmp_adaptation

The following functions are exported:

- `create_tables(Nodes) -> void()`
  [page 52] Creates the required Mnesia tables
- `name2index(Name) -> {ok, Index} | undefined`
  [page 52] Maps an event to the corresponding `eventIndex`
- `register_alarms(Alarms) -> void()`
  [page 52] Registers an alarm
- `register_events(Events) -> void()`
  [page 53] Registers an event
- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 53] Start the EVA SNMP adaptation service
- `alarmTable(Op, RowIndex, Cols) -> InstrumRet`
  [page 54] Instrumentation function for the alarmTable
- `curAlarmTable(Op, RowIndex, Cols) -> InstrumRet`
  [page 54] Instrumentation function for the currentAlarmTable
- `curAlarmLastTimeChanged(get) -> InstrumRet`
  [page 54] Instrumentation function for the currentAlarmTable
- `eventTable(Op, RowIndex, Cols) -> InstrumRet`
  [page 54] Instrumentation function for the eventTable

## log_snmp

The following functions are exported:

- `create_tables(Nodes) -> void()`
  [page 55] Create the Mnesia tables required by the log SNMP implementation
- `register_type(Type,TypeOid,TypeFunc)`
  [page 55] Register a log to the SNMP log functionality

- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 56] Start the LOG SNMP service
- `log_table(Op, RowIndex, Cols) -> InstrumRet`
  [page 56] Instrumentation function for the logTable
- `log_tr_table(Op, RowIndex, Cols) -> InstrumRet`
  [page 56] Instrumentation function for the logTransferTable

## log_snmpea

The following functions are exported:

- `snmpeaLogDiscriminator(Op,Val)`
  [page 57] Instrumentation function for snmpeaLogDiscriminator variable

# eva_log_snmp

Erlang Module

This module implements an SNMP interface to EVA LOG. The MIB implemented by this adaptation is OTP-EVA-LOG-MIB. The MIB is located in the directory `mibs` in the distribution.

## Exports

`create_tables(Nodes) -> void()`

Types:

- Nodes = [node()]

This function creates the necessary Mnesia tables for the eva log SNMP implementation. `Nodes` is a list of nodes where the tables should be replicated. This list of nodes should specify the same nodes where the application EVA can be run distributed, so that EVA can always have local access to the tables.

This function should be called once when installing the EVA application in the system.

`start_link(LogDir, MaxSize) -> {ok, Pid} | {error, Reason}`

Types:

- LogDir = string()
- MaxDirSize = integer()

Starts the EVA LOG SNMP implementation. This function can be used to include the service in a supervisor. Normally, functions in the supervisor `eva_sup` can be used instead.

`LogDir` is a directory where all manager created logs are stored. The directory must exist.

`MaxDirSize` is the maximum total space the logs manager created logs are allowed to use.

The function `create_tables/1` must be called before the server is started.

## Instrumentation Functions for the OTP-EVA-LOG-MIB

In some cases, other adaptations may need access to the SNMP specific data in EVA LOG. To do this, the instrumentation functions for the SNMP objects can be used. These instrumentation functions takes the arguments and return the values defined in the application `snmp`.

## Exports

`log_discr_table(Op, RowIndex, Cols) -> InstrumRet`

Instrumentation function for `logDiscriminatorTable`. This function assumes that access checks are made according to the MIB. It may crash if, for example, `logDiscrRowStatus` is set no `notReady`.

`log_table(Op, RowIndex, Cols) -> InstrumRet`

Instrumentation function for `logTable`. This function assumes that access checks are made according to the MIB. It may crash if, for example, `logOperStatus` is set.

`log_total_max_allowed(get) -> InstrumRet`

Instrumentation function for `logTotalMaxAllowedSize`.

`log_total_max_size(get) -> InstrumRet`

Instrumentation function for `logTotalMaxSize`.

## See Also

eva_log(3), eva_sup(3)

# eva_snmp_adaptation

Erlang Module

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

The resources generate events and alarms using the API provided by EVA. They are not aware that the events and alarms are sent as SNMP traps to SNMP managers.

However, each trap to be sent must be defined in an SNMP MIB, and there must be instrumentation functions that translate the EVA events and alarms into SNMP traps. Normally, each event and alarm in the system is mapped onto one separate SNMP trap. This mapping is done by registration of the events. The following functions are available for the registration. They could be called when the corresponding MIB is loaded.

## Exports

`create_tables(Nodes) -> void()`

Types:

- Nodes = [node()]

This function creates the Mnesia tables required for the `eva_snmp_adaptation`. `Nodes` is a list of nodes where the tables should be replicated. This list of nodes should specify the same nodes where the application EVA can be run distributed, so that EVA always can have local access to the tables.

This function should be called once when installing the EVA application in the system.

`name2index(Name) -> {ok, Index} | undefined`

Types:

- Name = atom()
- Index = integer()

Maps an event to the corresponding `eventIndex` value for the event, as defined in the `eventTable` in `OTP-EVA-MIB`.

`register_alarms(Alarms) -> void()`

Types:

- Alarms = [{Name, Trap, Treatment, Community, Func}]
- Name = Trap = atom()
- Func = fun(#alarm) -> {ok, ObjOID, CauseOID, SnmpVarbinds}
- Treatment = none | snmpTrap

- Community = string()
- ObjOid = CauseOid = [integer()]

This function must be used to register an EVA alarm as an SNMP alarm. It is used to associate each event with the corresponding SNMP trap and an Erlang instrumentation function which translates the `#alarm` into a trap. The corresponding `Trap` must be defined in an SNMP MIB. `Treatment` defines how this alarm is treated when it is generated. If it is `snmpTrap`, it is sent to the `Community`.

When the EVA alarm `Name` is generated by an application, the adaptation calls `Func(#alarm)`. The purpose of the `Func` is to translate the Erlang record `#alarm` into SNMP values. `ObjOID` is an OBJECT IDENTIFIER representation of the `#alarm.sender`, `CauseOID` is an OBJECT IDENTIFIER representation of the `#alarm.cause`, and `SnmpVarbinds` is a list of extra variable bindings for the trap. This list is as defined for `snmp:send_trap`.

The `alarm` record is defined in `eva.hrl`, available in the `include` directory in the distribution.

`register_events(Events) -> void()`

Types:

- Events = [{Name, Trap, Treatment, Community, Func}]
- Name = Trap = atom()
- Func = fun(#event) -> {ok, SnmpVarbinds}
- Treatment = none | snmpTrap
- Community = string()

This function must be used to register an EVA event as an SNMP event. It is used to associate each event with the corresponding SNMP trap and an Erlang instrumentation function which translates the `#event` into a trap. The corresponding `Trap` must be defined in an SNMP MIB. `Treatment` defines how this event is treated when it is generated. If it is `snmpTrap` it is sent to the `Community`.

When the EVA event `Name` is generated by an application, the adaptation calls `Func(#event)`. The purpose of the `Func` is to translate the Erlang record `#event` into SNMP values. The `SnmpVarbinds` is a list of extra variable bindings for the trap. This list is as defined for `snmp:send_trap`.

The `event` record is defined in `eva.hrl`, available in the `include` directory in the distribution.

`start_link() -> {ok, Pid} | {error, Reason}`

Starts the EVA SNMP adaptation. This function can be used to include the service in a supervisor. Normally, functions in the supervisor `eva_sup` can be used instead.

The function `create_tables/1` must be called before the service is started.

An EVA adaptation is always implemented as a `gen_event` handler. So is `eva_snmp_adaptation`. But in order to supervise this service from an ordinary supervisor, this function creates a process that supervises the `gen_event` handler.

## Instrumentation Functions for the OTP-EVA-MIB

In some cases, other adaptations may need access to the SNMP specific data in EVA. To do this, the instrumentation functions for the SNMP objects can be used. These instrumentation functions takes the arguments and return the values defined in the application `snmp`.

## Exports

`alarmTable(Op, RowIndex, Cols) -> InstrumRet`

Instrumentation function for `alarmTable`. This function assumes that access checks are made according to the MIB, so it may crash if, for example, `alarmClass` is set.

`curAlarmTable(Op, RowIndex, Cols) -> InstrumRet`

Instrumentation function for `currentAlarmTable`. This function assumes that access checks are made according to the MIB, so it may crash if, for example, `currentAlarmSeverity` is set.

`curAlarmLastTimeChanged(get) -> InstrumRet`

Instrumentation function for `currentAlarmLastTimeSent`.

`eventTable(Op, RowIndex, Cols) -> InstrumRet`

Instrumentation function for `eventTable`. This function assumes that access checks are made according to the MIB, so it may crash if, for example, `eventTrapName` is set.

## See Also

eva(3), eva_sup(3), gen_event(3), snmp(3)

# log_snmp

Erlang Module

This module implements an SNMP interface to the log service in the EVA application. The MIB implemented by this adaptation is OTP-LOG-MIB. The MIB is located in the directory `mibs` in the distribution.

## Exports

`create_tables(Nodes) -> void()`

Types:

- Nodes = [node()]

This function creates the Mnesia tables required by the log SNMP implementation. `Nodes` is a list of nodes where the tables should be replicated. This list of nodes should specify the same nodes where the application EVA can be run distributed, in order for the log server to always have local access to the tables.

This function should be called once when installing the EVA application in the system.

`register_type(Type,TypeOid,TypeFunc)`

Types:

- Type = term()
- TypeOid = oid() = [int()]
- TypeFunc = {M,F,A}
- M = F = atom()
- A = list()

This function is used to register a type of log to the SNMP log functionality. The `Type` is the same as the `Type` argument given to `log:open/3`.

The purpose of this function is to tell the SNMP LOG functions that all logs of type `Type` have an SNMP type defined in some MIB (`TypeOid`, defined as an OBJECT IDENTITY), and that the `TypeFunc` should be used to control creation and transfer of logs of this type.

The type control function (`TypeFunc`) will be called when a manger tries to create or delete a log of type `TypeOid`, or when he tries to transfer a log of this type. The purpose of this function is to check if creation is possible, and to format the log when it is transferred. The function should be defined as:

M:F(validate_creation, LogIndex, Cols | A) -> true | false | {SnmpErr, Col} Called when the manager tries to create a new log of type `TypeOid`. It is supposed to check if it is possible to create a new log of this type. If it is, it should return `true`. If it is never possible to create logs of this type, it should return `false`. Otherwise, the creation is not possible becasue some resource is not available, and the function should return `{SnmpError, Col}` (see definition of SNMP instrumenetation functions for a description of this).

M:F(create, Log | A) -> ok | error Called when `M:F(validate_creation, ...)` returned `true`. This function is supposed to create the log. `Log` is a `#log` record, defined in `log.hrl`.

M:F(delete, Log | A) -> void() Called when a log previously created by a manager is deleted.

M:F(search, LogIndex, LogTrIndex | A) -> SearchFunc Called when the manager activates a log transfer for a log of this type. The `LogIndex` is the index into `logTable`, and `{LogIndex, LogTrIndex}` is the index into the `logTransferTable`. This function is supposed to return a search function as specified in `log:transfer/5`. The records for these tables are defined in `include/log_snmp.hrl`.

`start_link() -> {ok, Pid} | {error, Reason}`

Starts the LOG SNMP implementation. This function can be used to include the service in a supervisor. Normally, functions in the supervisor `log_sup` can be used instead.

The function `create_tables/1` must have been called before the server is started.

## Instrumentation functions for the OTP-LOG-MIB

In some cases other adaptations may need access to the SNMP specific data in LOG. To do this, the instrumentation functions for the SNMP objects can be used. These instrumentation functions takes the arguments and return the values defined in the application `snmp`.

## Exports

`log_table(Op, RowIndex, Cols) -> InstrumRet`

Instrumentation function for `logTable`. This function assumes that access checks are made according to the MIB, so it may crash if e.g. `logOperStatus` is set.

`log_tr_table(Op, RowIndex, Cols) -> InstrumRet`

Instrumentation function for `logTransferTable`. This function assumes that access checks are made according to the MIB.

# log_snmpea

Erlang Module

This module contains the instrumentation functions for the OTP-SNMPEA-LOG-MIB.

In some cases other adaptations may need access to the SNMP specific data in EVA. To do this, the instrumentation functions for the SNMP objects can be used. These instrumentation functions takes the arguments and return the values defined in the application `snmp`.

## Exports

`snmpeaLogDiscriminator(Op,Val)`

Instrumentation function for the `snmpeaLogDiscriminator` variable.

# Glossary

## EVA adaptation

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

## NE

Network Element; In OTP, the Network Element is the entire distributed OTP system, meaning that the distributed OTP system is managed as one entity.

# Index of Modules and Functions

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