Simple Network Management Protocol (SNMP)

Simple Network Management Protocol (SNMP) 4.19

March 14 2011

March 14 2011

1 User's Guide

A multilingual Simple Network Management Protocol application, featuring an Extensible Agent, a simple manager and a MIB compiler and facilities for implementing SNMP MIBs etc.

1.1 SNMP Introduction

The SNMP development toolkit contains the following parts:

- An Extensible multi-lingual SNMP agent, which understands SNMPv1 (RFC1157), SNMPv2c (RFC1901, 1905, 1906 and 1907), SNMPv3 (RFC2271, 2272, 2273, 2274 and 2275), or any combination of these protocols.
- A multi-lingual SNMP manager.
- A MIB compiler, which understands SMIv1 (RFC1155, 1212, and 1215) and SMIv2 (RFC1902, 1903, and 1904).

The SNMP development tool provides an environment for rapid agent/manager prototyping and construction. With the following information provided, this tool is used to set up a running multi-lingual SNMP agent/manager:

- a description of a Management Information Base (MIB) in Abstract Syntax Notation One (ASN.1)
- instrumentation functions for the managed objects in the MIB, written in Erlang.

The advantage of using an extensible (agent/manager) toolkit is to remove details such as type-checking, access rights, Protocol Data Unit (PDU), encoding, decoding, and trap distribution from the programmer, who only has to write the instrumentation functions, which implement the MIBs. The `get-next` function only has to be implemented for tables, and not for every variable in the global naming tree. This information can be deduced from the ASN.1 file.

1.1.1 Scope and Purpose

This manual describes the SNMP development tool, as a component of the Erlang/Open Telecom Platform development environment. It is assumed that the reader is familiar with the Erlang Development Environment, which is described in a separate User's Guide.

1.1.2 Prerequisites

The following prerequisites is required for understanding the material in the SNMP User's Guide:

- the basics of the Simple Network Management Protocol version 1 (SNMPv1)
- the basics of the community-based Simple Network Management Protocol version 2 (SNMPv2c)
- the basics of the Simple Network Management Protocol version 3 (SNMPv3)
- the knowledge of defining MIBs using SMIv1 and SMIv2
- familiarity with the Erlang system and Erlang programming

The tool requires Erlang release 4.7 or later.

1.1.3 Definitions

The following definitions are used in the SNMP User's Guide.

MIB

The conceptual repository for management information is called the Management Information Base (MIB). It does not hold any data, merely a definition of what data can be accessed. A definition of an MIB is a description of a collection of managed objects.

SMI

The MIB is specified in an adapted subset of the Abstract Syntax Notation One (ASN.1) language. This adapted subset is called the Structure of Management Information (SMI).

ASN.1

ASN.1 is used in two different ways in SNMP. The SMI is based on ASN.1, and the messages in the protocol are defined by using ASN.1.

Managed object

A resource to be managed is represented by a managed object, which resides in the MIB. In an SNMP MIB, the managed objects are either:

- *scalar variables*, which have only one instance per context. They have single values, not multiple values like vectors or structures.
- *tables*, which can grow dynamically.
- a *table element*, which is a special type of scalar variable.

Operations

SNMP relies on the three basic operations: get (object), set (object, value) and get-next (object).

Instrumentation function

An instrumentation function is associated with each managed object. This is the function, which actually implements the operations and will be called by the agent when it receives a request from the management station.

Manager

A manager generates commands and receives notifications from agents. There usually are only a few managers in a system.

Agent

An agent responds to commands from the manager, and sends notification to the manager. There are potentially many agents in a system.

1.1.4 About This Manual

In addition to this introductory chapter, the SNMP User's Guide contains the following chapters:

- Chapter 2: "Functional Description" describes the features and operation of the SNMP development toolkit. It includes topics on Sub-agents and MIB loading, Internal MIBs, and Traps.
- Chapter 3: "The MIB Compiler" describes the features and the operation of the MIB compiler.
- Chapter 4: "Running the application" describes how to start and configure the application. Topics on how to debug the application are also included.
- Chapter 5: "Definition of Agent Configuration Files" is a reference chapter, which contains more detailed information about the agent configuration files.
- Chapter 6: "Definition of Manager Configuration Files" is a reference chapter, which contains more detailed information about the manager configuration files.
- Chapter 7: "Agent Implementation Example" describes how an MIB can be implemented with the SNMP Development Toolkit. Implementation examples are included.
- Chapter 8: "Instrumentation Functions" describes how instrumentation functions should be defined in Erlang for the different operations.
- Chapter 9: "Definition of Instrumentation Functions" is a reference chapter which contains more detailed information about the instrumentation functions.

- Chapter 10: "Definition of Agent Net if" is a reference chapter, which describes the Agent Net if function in detail.
- Chapter 11: "Definition of Manager Net if" is a reference chapter, which describes the Manager Net if function in detail.
- Chapter 12: "Advanced Agent Topics" describes sub-agents, agent semantics, audit trail logging, and the consideration of distributed tables.
- Appendix A describes the conversion of SNMPv2 to SNMPv1 error messages.
- Appendix B contains the RFC1903 text on `RowStatus`.

1.1.5 Where to Find More Information

Refer to the following documentation for more information about SNMP and about the Erlang/OTP development system:

- Marshall T. Rose (1991), "The Simple Book - An Introduction to Internet Management", Prentice-Hall
- Evan McGinnis and David Perkins (1997), "Understanding SNMP MIBs", Prentice-Hall
- RFC1155, 1157, 1212 and 1215 (SNMPv1)
- RFC1901-1907 (SNMPv2c)
- RFC1908, 2089 (coexistence between SNMPv1 and SNMPv2)
- RFC2271, RFC2273 (SNMP std MIBs)
- the Mnesia User's Guide
- the Erlang 4.4 Extensions User's Guide
- the Reference Manual
- the Erlang Embedded Systems User's Guide
- the System Architecture Support Libraries (SASL) User's Guide
- the Installation Guide
- the Asn1 User's Guide
- Concurrent Programming in Erlang, 2nd Edition (1996), Prentice-Hall, ISBN 0-13-508301-X.

1.2 Agent Functional Description

The SNMP agent system consists of one Master Agent and optional Sub-agents.

The tool makes it easy to dynamically extend an SNMP agent in run-time. MIBs can be loaded and unloaded at any time. It is also easy to change the implementation of an MIB in run-time, without having to recompile the MIB. The MIB implementation is clearly separated from the agent.

To facilitate incremental MIB implementation, the tool can generate a prototype implementation for a whole MIB, or parts thereof. This allows different MIBs and management applications to be developed at the same time.

1.2.1 Features

To implement an agent, the programmer writes instrumentation functions for the variables and the tables in the MIBs that the agent is going to support. A running prototype which handles `set`, `get`, and `get-next` can be created without any programming.

The toolkit provides the following:

- multi-lingual multi-threaded extensible SNMP agent
- easy writing of instrumentation functions with a high-level programming language
- basic fault handling such as automatic type checking

- access control
- authentication
- privacy through encryption
- loading and unloading of MIBs in run-time
- the ability to change instrumentation functions without recompiling the MIB
- rapid prototyping environment where the MIB compiler can use generic instrumentation functions, which later can be refined by the programmer
- a simple and extensible model for transaction handling and consistency checking of set-requests
- support of the sub-agent concept via distributed Erlang
- a mechanism for sending notifications (traps and informs)
- support for implementing SNMP tables in the Mnesia DBMS.

1.2.2 SNMPv1, SNMPv2 and SNMPv3

The SNMP development toolkit works with all three versions of Standard Internet Management Framework; SNMPv1, SNMPv2 and SNMPv3. They all share the same basic structure and components. And they follow the same architecture.

The versions are defined in following RFCs

- SNMPv1 RFC 1555, 1157 1212, 1213 and 1215
- SNMPv2 RFC 1902 - 1907
- SNMPv3 RFC 2570 - 2575

Over time, as the Framework has evolved from SNMPv1 , through SNMPv2, to SNMPv3 the definitions of each of these architectural components have become richer and more clearly defined, but the fundamental architecture has remained consistent.

The main features of SNMPv2 compared to SNMPv1 are:

- The `get-bulk` operation for transferring large amounts of data.
- Enhanced error codes.
- A more precise language for MIB specification

The standard documents that define SNMPv2 are incomplete, in the sense that they do not specify how an SNMPv2 message looks like. The message format and security issues are left to a special Administrative Framework. One such framework is the Community-based SNMPv2 Framework (SNMPv2c), which uses the same message format and framework as SNMPv1. Other experimental frameworks as exist, e.g. SNMPv2u and SNMPv2*.

The SNMPv3 specifications take a modular approach to SNMP. All modules are separated from each other, and can be extended or replaced individually. Examples of modules are Message definition, Security and Access Control. The main features of SNMPv3 are:

- Encryption and authentication is added.
- MIBs for agent configuration are defined.

All these specifications are commonly referred to as "SNMPv3", but it is actually only the Message module, which defines a new message format, and Security module, which takes care of encryption and authentication, that cannot be used with SNMPv1 or SNMPv2c. In this version of the agent toolkit, all the standard MIBs for agent configuration are used. This includes MIBs for definition of management targets for notifications. These MIBs are used regardless of which SNMP version the agent is configured to use.

The extensible agent in this toolkit understands the SNMPv1, SNMPv2c and SNMPv3. Recall that SNMP consists of two separate parts, the MIB definition language (SMI), and the protocol. On the protocol level, the agent can be configured to speak v1, v2c, v3 or any combination of them at the same time, i.e. a v1 request gets an v1 reply, a v2c request gets a v2c reply, and a v3 request gets a v3 reply. On the MIB level, the MIB compiler can compile both SMIv1 and SMIv2 MIBs. Once compiled, any of the formats can be loaded into the agent, regardless of which protocol version the agent is configured to use. This means that the agent translates from v2 notifications to v1 traps, and vice versa. For example, v2 MIBs can be loaded into an agent that speaks v1 only. The procedures for the translation between the two protocols are described in RFC 1908 and RFC 2089.

In order for an implementation to make full use of the enhanced SNMPv2 error codes, it is essential that the instrumentation functions always return SNMPv2 error codes, in case of error. These are translated into the corresponding SNMPv1 error codes by the agent, if necessary.

> Note:
>
> The translation from an SMIv1 MIB to an SNMPv2c or SNMPv3 reply is always very straightforward, but the translation from a v2 MIB to a v1 reply is somewhat more complicated. There is one data type in SMIv2, called `Counter64`, that an SNMPv1 manager cannot decode correctly. Therefore, an agent may never send a `Counter64` object to an SNMPv1 manager. The common practice in these situations is to simple ignore any `Counter64` objects, when sending a reply or a trap to an SNMPv1 manager. For example, if an SNMPv1 manager tries to GET an object of type `Counter64`, he will get a `noSuchName` error, while an SNMPv2 manager would get a correct value.

1.2.3 Operation

The following steps are needed to get a running agent:

- Write your MIB in SMI in a text file.
- Write the instrumentation functions in Erlang and compile them.
- Put their names in the association file.
- Run the MIB together with the association file through the MIB compiler.
- Configure the application (agent).
- Start the application (agent).
- Load the compiled MIB into the agent.

The figures in this section illustrate the steps involved in the development of an SNMP agent.

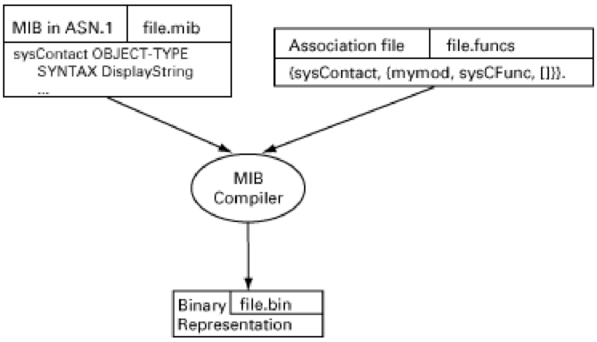
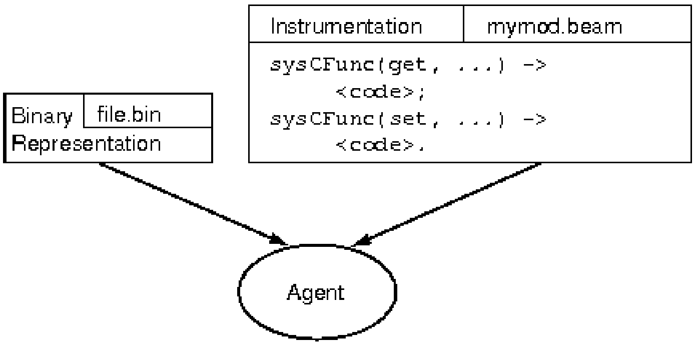


Figure 2.1: MIB Compiler Principles

The compiler parses the SMI file and associates each table or variable with an instrumentation function (see the figure *MIB Compiler Principles*). The actual instrumentation functions are not needed at MIB compile time, only their names.

The binary output file produced by the compiler is read by the agent at MIB load time (see the figure *Starting the Agent*). The instrumentation is ordinary Erlang code which is loaded explicitly or automatically the first time it is called.



```
sysCFunc(get, ...) ->
     <code>;
sysCFunc(set, ...) ->
     <code>.
```

Figure 2.2: Starting the Agent

The SNMP agent system consists of one Master Agent and optional sub-agents. The Master Agent can be seen as a special kind of sub-agent. It implements the core agent functionality, UDP packet processing, type checking, access control, trap distribution, and so on. From a user perspective, it is used as an ordinary sub-agent.

Sub-agents are only needed if your application requires special support for distribution from the SNMP toolkit. A sub-agent can also be used if the application requires a more complex set transaction scheme than is found in the master agent.

The following illustration shows how a system can look in runtime.

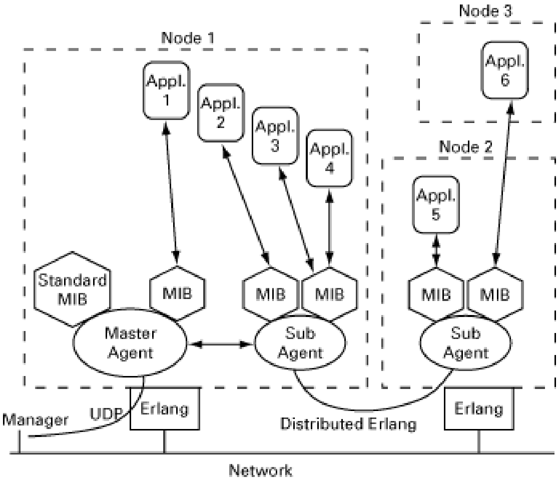


Figure 2.3: Architecture

A typical operation could include the following steps:

- The Manager sends a request to the Agent.
- The Master Agent decodes the incoming UDP packet.
- The Master Agent determines which items in the request that should be processed here and which items should be forwarded to its subagent.
- Step 3 is repeated by all subagents.
- Each sub-agent calls the instrumentation for its loaded MIBs.
- The results of calling the instrumentation are propagated back to the Master Agent.
- The answer to the request is encoded to a UDP Protocol Data Unit (PDU).

The sequence of steps shown is probably more complex than normal, but it illustrates the amount of functionality which is available. The following points should be noted:

- An agent can have many MIBs loaded at the same time.
- Sub-agents can also have sub-agents. Each sub-agent can have an arbitrary number of child sub-agents registered, forming a hierarchy.
- One MIB can communicate with many applications.
- Instrumentation can use Distributed Erlang to communicate with an application.

Most applications only need the Master Agent because an agent can have multiple MIBs loaded at the same time.

1.2.4 Sub-agents and MIB Loading

Since applications tend to be transient (they are dynamically loaded and unloaded), the management of these applications must be dynamic as well. For example, if we have an equipment MIB for a rack and different MIBs for boards, which can be installed in the rack, the MIB for a card should be loaded when the card is inserted, and unloaded when the card is removed.

In this agent system, there are two ways to dynamically install management information. The most common way is to load an MIB into an agent. The other way is to use a sub-agent, which is controlled by the application and is able to register and unregister itself. A sub-agent can register itself for managing a sub-tree (not to be mixed up with `erlang:register`). The sub-tree is identified by an Object Identifier. When a sub-agent is registered, it receives all requests for this particular sub-tree and it is responsible for answering them. It should also be noted that a sub-agent can be started and stopped at any time.

Compared to other SNMP agent packages, there is a significant difference in this way of using sub-agents. Other packages normally use sub-agents to load and unload MIBs in run-time. In Erlang, it is easy to load code in run-time and it is possible to load an MIB into an existing sub-agent. It is not necessary to create a new process for handling a new MIB.

Sub-agents are used for the following reasons:

- to provide a more complex set-transaction scheme than master agent
- to avoid unnecessary process communication
- to provide a more lightweight mechanism for loading and unloading MIBs in run-time
- to provide interaction with other SNMP agent toolkits.

Refer to the chapter *Advanced Agent Topics* in this User's Guide for more information about these topics.

The communication protocol between sub-agents is the normal message passing which is used in distributed Erlang systems. This implies that sub-agent communication is very efficient compared to SMUX, DPI, AgentX, and similar protocols.

1.2.5 Contexts and Communities

A context is a collection of management information accessible by an SNMP entity. An instance of a management object may exist in more than one context. An SNMP entity potentially has access to many contexts.

Each managed object can exist in many instances within a SNMP entity. To identify the instances, specified by an MIB module, a method to distinguish the actual instance by its 'scope' or context is used. Often the context is a physical or a logical device. It can include multiple devices, a subset of a single device or a subset of multiple devices, but the context is always defined as a subset of a single SNMP entity. To be able to identify a specific item of management information within an SNMP entity, the context, the object type and its instance must be used.

For example, the managed object type `ifDescr` from RFC1573, is defined as the description of a network interface. To identify the description of device-X's first network interface, four pieces of information are needed:

the snmpEngineID of the SNMP entity which provides access to the management information at device-X, the `contextName` (device-X), the managed object type (`ifDescr`), and the instance ("1").

In SNMPv1 and SNMPv2c, the community string in the message was used for (at least) three different purposes:

- to identify the context
- to provide authentication
- to identify a set of trap targets

In SNMPv3, each of these usage areas has its own unique mechanism. A context is identified by the name of the SNMP entity, `contextEngineID`, and the name of the context, `contextName`. Each SNMPv3 message contains values for these two parameters.

There is a MIB, SNMP-COMMUNITY-MIB, which maps a community string to a `contextEngineID` and `contextName`. Thus, each message, an SNMPv1, SNMPv2c or an SNMPv3 message, always uniquely identifies a context.

For an agent, the `contextEngineID` identified by a received message, is always equal to the `snmpEngineID` of the agent. Otherwise, the message was not intended for the agent. If the agent is configured with more than one context, the instrumentation code must be able to figure out for which context the request was intended. There is a function `snmpa:current_context/0` provided for this purpose.

By default, the agent has no knowledge of any other contexts than the default context, `""`. If it is to support more contexts, these must be explicitly added, by using an appropriate configuration file *Agent Configuration Files*.

1.2.6 Management of the Agent

There is a set of standard MIBs, which are used to control and configure an SNMP agent. All of these MIBs, with the exception of the optional SNMP-PROXY-MIB (which is only used for proxy agents), are implemented in this agent. Further, it is configurable which of these MIBs are actually loaded, and thus made visible to SNMP managers. For example, in a non-secure environment, it might be a good idea to not make MIBs that define access control visible. Note, the data the MIBs define is used internally in the agent, even if the MIBs not are loaded. This chapter describes these standard MIBs, and some aspects of their implementation.

Any SNMP agent must implement the `system` group and the `snmp` group, defined in MIB-II. The definitions of these groups have changed from SNMPv1 to SNMPv2. MIBs and implementations for both of these versions are Provided in the distribution. The MIB file for SNMPv1 is called STANDARD-MIB, and the corresponding for SNMPv2 is called SNMPv2-MIB. If the agent is configured for SNMPv1 only, the STANDARD-MIB is loaded by default; otherwise, the SNMPv2-MIB is loaded by default. It is possible to override this default behavior, by explicitly loading another version of this MIB, for example, you could choose to implement the union of all objects in these two MIBs.

An SNMPv3 agent must implement the SNMP-FRAMEWORK-MIB and SNMP-MPD-MIB. These MIBs are loaded by default, if the agent is configured for SNMPv3. These MIBs can be loaded for other versions as well.

There are five other standard MIBs, which also may be loaded into the agent. These MIBs are:

- SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB, which defines managed objects for configuration of management targets, i.e. receivers of notifications (traps and informs). These MIBs can be used with any SNMP version.
- SNMP-VIEW-BASED-ACM-MIB, which defined managed objects for access control. This MIB can be used with any SNMP version.
- SNMP-COMMUNITY-MIB, which defines managed objects for coexistence of SNMPv1 and SNMPv2c with SNMPv3. This MIB is only useful if SNMPv1 or SNMPv2c is used, possibly in combination with SNMPv3.
- SNMP-USER-BASED-SM-MIB, which defines managed objects for authentication and privacy. This MIB is only useful with SNMPv3.

All of these MIBs should be loaded into the Master Agent. Once loaded, these MIBs are always available in all contexts.

The ASN.1 code, the Erlang source code, and the generated `.hrl` files for them are provided in the distribution and are placed in the directories `mibs`, `src`, and `include`, respectively, in the `snmp` application.

The `.hrl` files are generated with `snmpc:mib_to_hrl/1`. Include these files in your code as in the following example:

```
-include_lib("snmp/include/SNMPv2-MIB.hrl").
```

The initial values for the managed objects defined in these tables, are read at start-up from a set of configuration files. These are described in *Configuration Files*.

STANDARD-MIB and SNMPv2-MIB

These MIBs contain the `snmp-` and `system` groups from MIB-II which is defined in RFC1213 (STANDARD-MIB) or RFC1907 (SNMPv2-MIB). They are implemented in the `snmp_standard_mib` module. The `snmp` counters all reside in volatile memory and the `system` and `snmpEnableAuthenTraps` variables in persistent memory, using the SNMP built-in database (refer to the Reference Manual, section `snmp`, module `snmpa_local_db` for more details).

If another implementation of any of these variables is needed, e.g. to store the persistent variables in a Mnesia database, an own implementation of the variables must be made. That MIB will be compiled and loaded instead of the default MIB. The new compiled MIB must have the same name as the original MIB (i.e. STANDARD-MIB or SNMPv2-MIB), and be located in the SNMP configuration directory (see *Configuration Files*.)

One of these MIBs is always loaded. If only SNMPv1 is used, STANDARD-MIB is loaded, otherwise SNMPv2-MIB is loaded.

Data Types

There are some new data types in SNMPv2 that are useful in SNMPv1 as well. In the STANDARD-MIB, three data types are defined, `RowStatus`, `TruthValue` and `DateAndTime`. These data types are originally defined as textual conventions in SNMPv2-TC (RFC1903).

SNMP-FRAMEWORK-MIB and SNMP-MPD-MIB

The SNMP-FRAMEWORK-MIB and SNMP-MPD-MIB define additional read-only managed objects, which is used in the generic SNMP framework defined in RFC2271 and the generic message processing and dispatching module defined in RFC2272. They are generic in the sense that they are not tied to any specific SNMP version.

The objects in these MIBs are implemented in the modules `snmp_framework_mib` and `snmp_standard_mib`, respectively. All objects reside in volatile memory, and the configuration files are always reread at start-up.

If SNMPv3 is used, these MIBs are loaded by default.

SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB

The SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB define managed objects for configuration of notification receivers. They are described in detail in RFC2273. Only a brief description is given here.

All tables in these MIBs have a column of type `StorageType`. The value of this column specifies how each row is stored, and what happens in case of a restart of the agent. The implementation supports the values `volatile` and `nonVolatile`. When the tables are initially filled with data from the configuration files, these rows will automatically have storage type `nonVolatile`. Should the agent restart, all `nonVolatile` rows survive the restart, while the `volatile` rows are lost. The configuration files are not read at restart, by default.

These MIBs are not loaded by default.

snmpNotifyTable

An entry in the `snmpNotifyTable` selects a set of management targets, which should receive notifications, as well as the type (trap or inform) of notification that should be sent to each selected management target. When an application sends a notification using the function `send_notification/5` or the function `send_trap` the parameter `NotifyName`, specified in the call, is used as an index in the table. The notification is sent to the management targets selected by that entry.

snmpTargetAddrTable

An entry in the `snmpTargetAddrTable` defines transport parameters (such as IP address and UDP port) for each management target. Each row in the `snmpNotifyTable` refers to potentially many rows in the `snmpTargetAddrTable`. Each row in the `snmpTargetAddrTable` refers to an entry in the `snmpTargetParamsTable`.

snmpTargetParamsTable

An entry in the `snmpTargetParamsTable` defines which SNMP version to use, and which security parameters to use.

Which SNMP version to use is implicitly defined by specifying the Message Processing Model. This version of the agent handles the models `v1`, `v2c` and `v3`.

Each row specifies which security model to use, along with security level and security parameters.

SNMP-VIEW-BASED-ACM-MIB

The SNMP-VIEW-BASED-ACM-MIB defines managed objects to control access to the the managed objects for the managers. The View Based Access Control Module (VACM) can be used with any SNMP version. However, if it is used with SNMPv1 or SNMPv2c, the SNMP-COMMUNITY-MIB defines additional objects to map community strings to VACM parameters.

All tables in this MIB have a column of type `StorageType`. The value of this column specifies how each row is stored, and what happens in case of a restart of the agent. The implementation supports the values `volatile` and `nonVolatile`. When the tables are initially filled with data from the configuration files, these rows will automatically have storage type `nonVolatile`. Should the agent restart, all `nonVolatile` rows survive the restart, while the `volatile` rows are lost. The configuration files are not read at restart by default.

This MIB is not loaded by default.

VACM is described in detail in RFC2275. Here is only a brief description given.

The basic concept is that of a *MIB view*. An MIB view is a subset of all the objects implemented by an agent. A manager has access to a certain MIB view, depending on which security parameters are used, in which context the request is made, and which type of request is made.

The following picture gives an overview of the mechanism to select an MIB view:

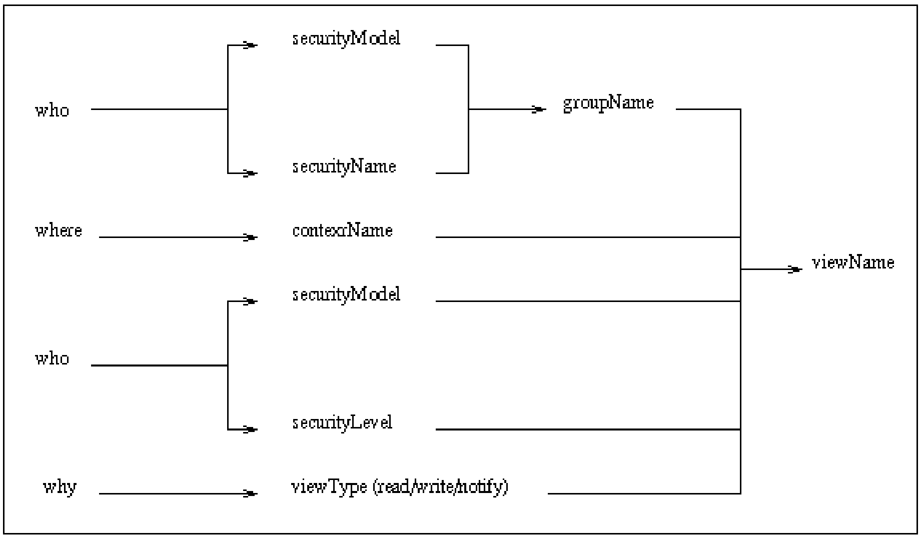


Figure 2.4: Overview of the mechanism of MIB selection

vacmContextTable

The `vacmContextTable` is a read-only table that lists all available contexts.

vacmSecurityToGroupTable

The `vacmSecurityToGroupTable` maps a `securityModel` and a `securityName` to a `groupName`.

vacmAccessTable

The `vacmAccessTable` maps the `groupName` (found in `vacmSecurityToGroupTable`), `contextName`, `securityModel`, and `securityLevel` to an MIB view for each type of operation (read, write, or notify). The MIB view is represented as a `viewName`. The definition of the MIB view represented by the `viewName` is found in the `vacmViewTreeFamilyTable`

vacmViewTreeFamilyTable

The `vacmViewTreeFamilyTable` is indexed by the `viewName`, and defines which objects are included in the MIB view.

The MIB definition for the table looks as follows:

```
VacmViewTreeFamilyEntry ::= SEQUENCE
    {
        vacmViewTreeFamilyViewName     SnmpAdminString,
        vacmViewTreeFamilySubtree      OBJECT IDENTIFIER,
        vacmViewTreeFamilyMask         OCTET STRING,
        vacmViewTreeFamilyType         INTEGER,
        vacmViewTreeFamilyStorageType  StorageType,
        vacmViewTreeFamilyStatus       RowStatus
    }

INDEX { vacmViewTreeFamilyViewName,
        vacmViewTreeFamilySubtree
      }
```

Each `vacmViewTreeFamilyViewName` refers to a collection of sub-trees.

MIB View Semantics

An MIB view is a collection of included and excluded sub-trees. A sub-tree is identified by an OBJECT IDENTIFIER. A mask is associated with each sub-tree.

For each possible MIB object instance, the instance belongs to a sub-tree if:

- the OBJECT IDENTIFIER name of that MIB object instance comprises at least as many sub-identifiers as does the sub-tree, and
- each sub-identifier in the name of that MIB object instance matches the corresponding sub-identifier of the sub-tree whenever the corresponding bit of the associated mask is 1 (0 is a wild card that matches anything).

Membership of an object instance in an MIB view is determined by the following algorithm:

- If an MIB object instance does not belong to any of the relevant sub-trees, then the instance is not in the MIB view.
- If an MIB object instance belongs to exactly one sub-tree, then the instance is included in, or excluded from, the relevant MIB view according to the type of that entry.
- If an MIB object instance belongs to more than one sub-tree, then the sub-tree which comprises the greatest number of sub-identifiers, and is the lexicographically greatest, is used.

> **Note:**
>
> If the OBJECT IDENTIFIER is longer than an OBJECT IDENTIFIER of an object type in the MIB, it refers to object instances. Because of this, it is possible to control whether or not particular rows in a table shall be visible.

SNMP-COMMUNITY-MIB

The SNMP-COMMUNITY-MIB defines managed objects that is used for coexistence between SNMPv1 and SNMPv2c with SNMPv3. Specifically, it contains objects for mapping between community strings and version-independent SNMP message parameters. In addition, this MIB provides a mechanism for performing source address validation on incoming requests, and for selecting community strings based on target addresses for outgoing notifications.

All tables in this MIB have a column of type `StorageType`. The value of this column specifies how each row is stored, and what happens in case of a restart of the agent. The implementation supports the values `volatile` and `nonVolatile`. When the tables are initially filled with data from the configuration files, these rows will automatically have storage type `nonVolatile`. Should the agent restart, all `nonVolatile` rows survive the restart, while the `volatile` rows are lost. The configuration files are not read at restart, by default.

This MIB is not loaded by default.

SNMP-USER-BASED-SM-MIB

The SNMP-USER-BASED-SM-MIB defines managed objects that is used for the User-Based Security Model.

All tables in this MIB have a column of type `StorageType`. The value of the column specifies how each row is stored, and what happens in case of a restart of the agent. The implementation supports the values `volatile` and `nonVolatile`. When the tables are initially filled with data from the configuration files, these rows will automatically have storage type `nonVolatile`. Should the agent restart, all `nonVolatile` rows survive the restart, while the `volatile` rows are lost. The configuration files are not read at restart, by default.

This MIB is not loaded by default.

OTP-SNMPEA-MIB

The OTP-SNMPEA-MIB was used in earlier versions of the agent, before standard MIBs existed for access control, MIB views, and trap target specification. All objects in this MIB are now obsolete.

1.2.7 Notifications

Notifications are defined in SMIv1 with the TRAP-TYPE macro in the definition of an MIB (see RFC1215). The corresponding macro in SMIv2 is NOTIFICATION-TYPE. When an application decides to send a notification, it calls one of the following functions:

```
snmpa:send_notification(Agent, Notification, Receiver
                        [, NotifyName, ContextName, Varbinds])
snmpa:send_trap(Agent, Notification, Community [, Receiver, Varbinds])
```

providing the registered name or process identifier of the agent where the MIB, which defines the notification is loaded and the symbolic name of the notification.

If the `send_notification/3,4` function is used, all management targets are selected, as defined in RFC2273. The `Receiver` parameter defines where the agent should send information about the delivery of inform requests.

If the `send_notification/5` function is used, an `NotifyName` must be provided. This parameter is used as an index in the `snmpNotifyTable`, and the management targets defined by that single entry is used.

The `send_notification/6` function is the most general version of the function. A `ContextName` must be specified, from which the notification will be sent. If this parameter is not specified, the default context (`""`) is used.

The function `send_trap` is kept for backwards compatibility and should not be used in new code. Applications that use this function will continue to work. The `snmpNotifyName` is used as the community string by the agent when a notification is sent.

Notification Sending

The simplest way to send a notification is to call the function `snmpa:send_notification(Agent, Notification, no_receiver)`. In this case, the agent performs a get-operation to retrieve the object values that are defined in the notification specification (with the TRAP-TYPE or NOTIFICATION-TYPE macros). The notification is sent to all managers defined in the target and notify tables, either unacknowledged as traps, or acknowledged as inform requests.

If the caller of the function wants to know whether or not acknowledgments are received for a certain notification (provided it is sent as an inform), the `Receiver` parameter can be specified as `{Tag, ProcessName}` (refer to the Reference Manual, section snmp, module `snmp` for more details). In this case, the agent send a message `{snmp_notification, Tag, {got_response, ManagerAddr}}` or `{snmp_notification, Tag, {no_response, ManagerAddr}}` for each management target.

Sometimes it is not possible to retrieve the values for some of the objects in the notification specification with a get-operation. However, they are known when the `send_notification` function is called. This is the case if an object is an element in a table. It is possible to give the values of some objects to the `send_notification` function `snmpa:send_notification(Agent, Notification, Receiver, Varbinds)`. In this function, `Varbinds` is a list of `Varbind`, where each `Varbind` is one of:

- `{Variable, Value}`, where `Variable` is the symbolic name of a scalar variable referred to in the notification specification.
- `{Column, RowIndex, Value}`, where `Column` is the symbolic name of a column variable. `RowIndex` is a list of indices for the specified element. If this is the case, the OBJECT IDENTIFIER sent in the trap is the

`RowIndex` appended to the OBJECT IDENTIFIER for the table column. This is the OBJECT IDENTIFIER which specifies the element.

- `{OID, Value}`, where `OID` is the OBJECT IDENTIFIER for an instance of an object, scalar variable or column variable.

For example, to specify that `sysLocation` should have the value `"upstairs"` in the notification, we could use one of:

- `{sysLocation, "upstairs"}` or
- `{[1,3,6,1,2,1,1,6,0], "upstairs"}`

It is also possible to specify names and values for extra variables that should be sent in the notification, but were not defined in the notification specification.

The notification is sent to all management targets found in the tables. However, make sure that each manager has access to the variables in the notification. If a variable is outside a manager's MIB view, this manager will not receive the notification.

> Note:
>
> By definition, it is not possible to send objects with ACCESS `not-accessible` in notifications. However, historically this is often done and for this reason we allow it in notification sending. If a variable has ACCESS `not-accessible`, the user must provide a value for the variable in the `Varbinds` list. It is not possible for the agent to perform a get-operation to retrieve this value.

Notification Filters

It is possible to add *notification filters* to an agent. These filters will be called when a notification is to be sent. Their purpose is to allow modification, suppression or other type of actions.

A notification filter is a module implementing the *snmpa_notification_filter* behaviour. A filter is added/deleted using the functions: *snmpa:register_notification_filter* and *snmpa:unregister_notification_filter*.

Unless otherwise specified, the order of the registered filters will be the order in which they are registered.

Sub-agent Path

If a value for an object is not given to the `send_notification` function, the sub-agent will perform a get-operation to retrieve it. If the object is not implemented in this sub-agent, its parent agent tries to perform a get-operation to retrieve it. If the object is not implemented in this agent either, it forwards the object to its parent, and so on. Eventually the Master Agent is reached and at this point all unknown object values must be resolved. If some object is unknown even to the Master Agent, this is regarded as an error and is reported with a call to `user_err/2` of the error report module. No notifications are sent in this case.

For a given notification, the variables, which are referred to in the notification specification, must be implemented by the agent that has the MIB loaded, or by some parent to this agent. If not, the application must provide values for the unknown variables. The application must also provide values for all elements in tables.

1.2.8 Discovery

The *sender* is *authoritative* for messages containing payload which does *not* expect a response (for example SNMPv2-Trap, Response or Report PDU).

The *receiver* is *authoritative* for messages containing payload which expects a response (for example Get, GetNext, Get-Bulk, Set or Inform PDU).

The agent can both perform and respond to discovery.

The agent responds to discovery autonomously, without interaction by the user.

Initiating discovery towards a manager is done by calling the *discovery* function. The `EngineId` field of the target (manager) entry in the *target_addr.conf* file has to have the value `discovery`. Note that if the manager does not respond, the `Timeout` and `RetryCount` fields decide how long the function will hang before it returns.

Discovery can only be performed towards one manager at a time.

1.3 Manager Functional Description

1.3.1 Features

The manager provided with the tool is a lightweight manager that basically provides a means to communicate with agents.

It does not really implement any management capabilities by itself. That is up to the *user*.

A *user* in this context is basically a module implementing the *snmpm_user* behaviour. A *user* can issue snmp requests and receive notification/traps.

Agents to be accessed by the manager needs to be registered by a user. Once registered, they can be accessed by all registered users.

Notifications/traps from an agent is delivered to the user that did the registration.

Any message from an agent that is not registered is delivered to the *default user*.

By default, the *default user* is set to the `snmpm_user_default` module, which simply sends an info message to the error_logger. It is however highly recommended that this module be replaced by another that does something useful (see *configuration params* for more info).

When using version 3, then (at least one) *usm user* has to be registered.

Requests can be issued in two different ways. Synchronous (see *sync_set*, *sync_get*, *sync_get_next* and *sync_get_bulk*) and asynchronous (see *async_set*, *async_get*, *async_get_next* and *async_get_bulk*). With synchronous the snmp reply is returned by the function. With asynchronous, the reply will instead be delivered through a call to one of the `handle_pdu` callback function defined by the *handle_pdu* behaviour.

1.3.2 Operation

The following steps are needed to get the manager running:

- [optional] Implement the default user.
- Implement the user(s).
- Configure the application (manager).
- Start the application (manager).
- Register the user(s).
- The user(s) register their agents.

1.3.3 MIB loading

It is possible to load mibs into the manager, but this is not necessary for normal operation, and not recommended.

1.4 The MIB Compiler

The chapter *The MIB Compiler* describes the MIB compiler and contains the following topics:

- Operation

- Import
- Consistency checking between MIBs
- .hrl file generation
- Emacs integration
- Deviations from the standard

> **Note:**
>
> When importing MIBs, ensure that the imported MIBs as well as the importing MIB are compiled using the same version of the SNMP-compiler.

1.4.1 Operation

The MIB must be written as a text file in SMIv1 or SMIv2 using an ASN.1 notation before it will be compiled. This text file must have the same name as the MIB, but with the suffix `.mib`. This is necessary for handling the `IMPORT` statement.

The association file, which contains the names of instrumentation functions for the MIB, should have the suffix `.funcs`. If the compiler does not find the association file, it gives a warning message and uses default instrumentation functions. (See *Default Instrumentation* for more details).

The MIB compiler is started with a call to `snmpc:compile(<mibname>).` For example:

```
snmpc:compile("RFC1213-MIB").
```

The output is a new file which is called `<mibname>.bin`.

The MIB compiler understands both SMIv1 and SMIv2 MIBs. It uses the MODULE-IDENTITY statement to determinate if the MIB is written in SMI version 1 or 2.

1.4.2 Importing MIBs

The compiler handles the `IMPORT` statement. It is important to import the compiled file and not the ASN.1 (source) file. A MIB must be recompiled to make changes visible to other MIBs importing it.

The compiled files of the imported MIBs must be present in the current directory, or a directory in the current path. The path is supplied with the `{i, Path}` option, for example:

```
snmpc:compile("MY-MIB",
       [{i, ["friend_mibs/", "../standard_mibs/"]}]).
```

It is also possible to import MIBs from OTP applications in an `"include_lib"` like fashion with the `il` option. Example:

```
snmpc:compile("MY-MIB",
       [{il, ["snmp/priv/mibs/", "myapp/priv/mibs/"]}]).
```

finds the latest version of the snmp and myapp applications in the OTP system and uses the expanded paths as include paths.

Note that an SMIv2 MIB can import an SMIv1 MIB and vice versa.

The following MIBs are built-ins of the Erlang SNMP compiler: SNMPv2-SMI, RFC-1215, RFC-1212, SNMPv2-TC, SNMPv2-CONF, and RFC1155-SMI. They cannot therefore be compiled separately.

1.4.3 MIB Consistency Checking

When an MIB is compiled, the compiler detects if several managed objects use the same OBJECT IDENTIFIER. If that is the case, it issues an error message. However, the compiler cannot detect Oid conflicts between different MIBs. These kinds of conflicts generate an error at load time. To avoid this, the following function can be used to do consistency checking between MIBs:

```
erl>snmpc:is_consistent(ListOfMibNames).
```

ListOfMibNames is a list of compiled MIBs, for example ["RFC1213-MIB", "MY-MIB"]. The function also performs consistency checking of trap definitions.

1.4.4 .hrl File Generation

It is possible to generate an .hrl file which contains definitions of Erlang constants from a compiled MIB file. This file can then be included in Erlang source code. The file will contain constants for:

- object Identifiers for tables, table entries and variables
- column numbers
- enumerated values
- default values for variables and table columns.

Use the following command to generate a .hrl file from an MIB:

```
erl>snmpc:mib_to_hrl(MibName).
```

1.4.5 Emacs Integration

With the Emacs editor, the next-error (C-X `) function can be used indicate where a compilation error occurred, provided the error message is described by a line number.

Use M-x compile to compile an MIB from inside Emacs, and enter:

```
 erl -s snmpc compile <MibName> -noshell
```

An example of <MibName> is RFC1213-MIB.

1.4.6 Compiling from a Shell or a Makefile

The erlc commands can be used to compile SNMP MIBs. Example:

```
erlc MY-MIB.mib
```

All the standard `erlc` flags are supported, e.g.

```
erlc -I mymibs -o mymibs -W MY-MIB.mib
```

The flags specific to the MIB compiler can be specified by using the + syntax:

```
erlc +'{group_check,false}' MY-MIB.mib
```

1.4.7 Deviations from the Standard

In some aspects the Erlang MIB compiler does not follow or implement the SMI fully. Here are the differences:

- Tables must be written in the following order: `tableObject`, `entryObject`, `column1`, ..., `columnN` (in order).
- Integer values, for example in the `SIZE` expression must be entered in decimal syntax, not in hex or bit syntax.
- Symbolic names must be unique within a MIB and within a system.
- Hyphens are allowed in SMIv2 (a pragmatic approach). The reason for this is that according to SMIv2, hyphens are allowed for objects converted from SMIv1, but not for others. This is impossible to check for the compiler.
- If a word is a keyword in any of SMIv1 or SMIv2, it is a keyword in the compiler (deviates from SMIv1 only).
- Indexes in a table must be objects, not types (deviates from SMIv1 only).
- A subset of all semantic checks on types are implemented. For example, strictly the `TimeTicks` may not be sub-classed but the compiler allows this (standard MIBs must pass through the compiler) (deviates from SMIv2 only).
- The `MIB.Object` syntax is not implemented (since all objects must be unique anyway).
- Two different names cannot define the same OBJECT IDENTIFIER.
- The type checking in the SEQUENCE construct is non-strict (i.e. subtypes may be specified). The reason for this is that some standard MIBs use this.
- A definition has normally a status field. When the status field has the value deprecated, then the MIB-compiler will ignore this definition. With the MIB-compiler option `{deprecated,true}` the MIB-compiler does not ignore the deprecated definitions.
- An object has a DESCRIPTIONS field. The descriptions-field will not be included in the compiled mib by default. In order to get the description, the mib must be compiled with the option `description`.

1.5 Running the application

The chapter *Running the application* describes how the application is configured and started. The topics include:

- configuration directories and parameters
- modifying the configuration files
- starting the application (agent and/or manager)
- debugging the application (agent and/or manager)

Refer also to the chapter(s) *Definition of Agent Configuration Files* and *Definition of Manager Configuration Files* which contains more detailed information about the agent and manager configuration files.

1.5.1 Configuring the application

The following two directories must exist in the system to run the agent:

- the *configuration directory* stores all configuration files used by the agent (refer to the chapter *Definition of Agent Configuration Files* for more information).
- the *database directory* stores the internal database files.

The following directory must exist in the system to run the manager:

- the *configuration directory* stores all configuration files used by the manager (refer to the chapter *Definition of Manager Configuration Files* for more information).
- the *database directory* stores the internal database files.

The agent and manager uses (application) configuration parameters to find out where these directories are located. The parameters should be defined in an Erlang system configuration file. The following configuration parameters are defined for the SNMP application:

```
    agent_options() = [agent_option()]
    agent_option() = {restart_type,     restart_type()}     |
                     {agent_type,       agent_type()}       |
                     {agent_verbosity,  verbosity()}        |
                     {versions,         versions()}         |
                     {discovery,        agent_discovery()}  |
                     {priority,         priority()}         |
                     {multi_threaded,   multi_threaded()}   |
                     {db_dir,           db_dir()}           |
                     {db_init_error,    db_init_error()}    |
                     {local_db,         local_db()}         |
                     {net_if,           agent_net_if()}     |
                     {mibs,             mibs()}             |
                     {mib_storage,      mib_storage()}      |
                     {mib_server,       mib_server()}       |
                     {audit_trail_log,  audit_trail_log()}  |
                     {error_report_mod, error_report_mod()} |
                     {note_store,       note_store()}       |
                     {symbolic_store,   symbolic_store()}   |
                     {target_cache,     target_cache()}     |
                     {config,           agent_config()}
    manager_options() = [manager_option()]
    manager_option() = {restart_type,             restart_type()}    |
                       {net_if,                   manager_net_if()}  |
                       {server,                   server()}          |
                       {note_store,               note_store()}      |
                       {config,                   manager_config()}  |
                       {inform_request_behaviour, manager_irb()}     |
                       {mibs,                     manager_mibs()}    |
                       {priority,                 priority()}        |
                       {audit_trail_log,          audit_trail_log()} |
                       {versions,                 versions()}        |
                       {def_user_mod,             def_user_module()  |
                       {def_user_data,            def_user_data()}
```

Agent specific config options and types:

`agent_type() = master | sub <optional>`

If `master`, one master agent is started. Otherwise, no agents are started.

Default is `master`.

`agent_discovery() = [agent_discovery_opt()] <optional>`

`agent_discovery_opt() = {terminating, agent_terminating_discovery_opts()} | {originating, agent_originating_discovery_opts()}`

The `terminating` options effects discovery initiated by a manager.

The `originating` options effects discovery initiated by this agent.

For defaults see the options in `agent_discovery_opt()`.

`agent_terminating_discovery_opts() = [agent_terminating_discovery_opt()] <optional>`

`agent_terminating_discovery_opt() = {enable, boolean()} | {stage2, discovery | plain} | {trigger_username, string()}`

These are options effecting discovery `terminating` in this agent (i.e. initiated by a manager).

The default values for the `terminating` discovery options are:

- enable: `true`
- stage2: `discovery`
- trigger_username: `""`

`agent_originating_discovery_opts() = [agent_originating_discovery_opt()] <optional>`

`agent_originating_discovery_opt() = {enable, boolean()}`

These are options effecting discovery `originating` in this agent.

The default values for the `originating` discovery options are:

- enable: `true`

`multi_threaded() = bool() <optional>`

If `true`, the agent is multi-threaded, with one thread for each get request.

Default is `false`.

`db_dir() = string() <mandatory>`

Defines where the SNMP agent internal db files are stored.

`local_db() = [local_db_opt()] <optional>`

`local_db_opt() = {repair, agent_repair()} | {auto_save, agent_auto_save()} | {verbosity, verbosity()}`

Defines options specific for the SNMP agent local database.

For defaults see the options in `local_db_opt()`.

`agent_repair() = false | true | force <optional>`

When starting snmpa_local_db it always tries to open an existing database. If `false`, and some errors occur, a new database is created instead. If `true`, an existing file will be repaired. If `force`, the table will be repaired even if it was properly closed.

Default is `true`.

`agent_auto_save() = integer() | infinity <optional>`

The auto save interval. The table is flushed to disk whenever not accessed for this amount of time.

Default is `5000`.

`agent_net_if() = [agent_net_if_opt()] <optional>`

`agent_net_if_option() = {module, agent_net_if_module()} | {verbosity, verbosity()} | {options, agent_net_if_options()}`

Defines options specific for the SNMP agent network interface entity.

For defaults see the options in `agent_net_if_opt()`.

`agent_net_if_module() = atom() <optional>`

Module which handles the network interface part for the SNMP agent. Must implement the *snmpa_network_interface* behaviour.

Default is `snmpa_net_if`.

`agent_net_if_options() = [agent_net_if_option()] <optional>`

`agent_net_if_option() = {bind_to, bind_to()} | {sndbuf, sndbuf()} | {recbuf, recbuf()} | {no_reuse, no_reuse()} | {req_limit, req_limit()} | {filter, agent_net_if_filter_options()}`

These options are actually specific to the used module. The ones shown here are applicable to the default `agent_net_if_module()`.

For defaults see the options in `agent_net_if_option()`.

`agent_net_if_filter_options() = [agent_net_if_filter_option()] <optional>`

`agent_net_if_filter_option() = {module, agent_net_if_filter_module()}`

These options are actually specific to the used module. The ones shown here are applicable to the default `agent_net_if_filter_module()`.

For defaults see the options in `agent_net_if_filter_option()`.

`agent_net_if_filter_module() = atom() <optional>`

Module which handles the network interface filter part for the SNMP agent. Must implement the *snmpa_network_interface_filter* behaviour.

Default is `snmpa_net_if_filter`.

`req_limit() = integer() | infinity <optional>`

Max number of simultaneous requests handled by the agent.

Default is `infinity`.

`agent_mibs() = [string()] <optional>`

Specifies a list of MIBs (including path) that defines which MIBs are initially loaded into the SNMP master agent.

Note that the following will always be loaded:

- version v1: `STANDARD-MIB`
- version v2: `SNMPv2`
- version v3: `SNMPv2`, `SNMP-FRAMEWORK-MIB` and `SNMP-MPD-MIB`

Default is `[]`.

`mib_storage() = ets | {ets, Dir} | {ets, Dir, Action} | dets | {dets, Dir} | {dets, Dir, Action} | mnesia | {mnesia, Nodes} | {mnesia, Nodes, Action} <optional>`

Specifies how info retrieved from the mibs will be stored.

If mib_storage is {ets, Dir}, the table will also be stored on file. If Dir is default, then db_dir will be used.

If mib_storage is dets or if Dir is default, then db_dir will be used for Dir.

If mib_storage is mnesia then erlang:nodes() will be used for Nodes.

Default is ets.

Dir = default | string(). Dir is the directory where the files will be stored. If default, then db_dir will be used.

Nodes = visible | connected | [node()]. Nodes = visible is translated to erlang:nodes(visible). Nodes = connected is translated to erlang:nodes(connected). If Nodes = [] then the own node is assumed.

Action = clear | keep. Default is keep. Action is used to specify what shall be done if the mnesia/dets table already exist.

mib_server() = [mib_server_opt()] <optional>

mib_server_opt() = {mibentry_override, mibentry_override()} | {trapentry_override, trapentry_override()} | {verbosity, verbosity()} | {cache, mibs_cache()}

Defines options specific for the SNMP agent mib server.

For defaults see the options in mib_server_opt().

mibentry_override() = bool() <optional>

If this value is false, then when loading a mib each mib- entry is checked prior to installation of the mib. The purpose of the check is to prevent that the same symbolic mibentry name is used for different oid's.

Default is false.

trapentry_override() = bool() <optional>

If this value is false, then when loading a mib each trap is checked prior to installation of the mib. The purpose of the check is to prevent that the same symbolic trap name is used for different trap's.

Default is false.

mibs_cache() = bool() | mibs_cache_opts() <optional>

Shall the agent utilize the mib server lookup cache or not.

Default is true (in which case the mibs_cache_opts() default values apply).

mibs_cache_opts() = [mibs_cache_opt()] <optional>

mibs_cache_opt() = {autogc, mibs_cache_autogc()} | {gclimit, mibs_cache_gclimit()} | {age, mibs_cache_age()}

Defines options specific for the SNMP agent mib server cache.

For defaults see the options in mibs_cache_opt().

mibs_cache_autogc() = bool() <optional>

Defines if the mib server shall perform cache gc automatically or leave it to the user (see *gc_mibs_cache/0,1,2,3*).

Default is true.

mibs_cache_age() = integer() > 0 <optional>

Defines how old the entries in the cache will be allowed before they are GC'ed (assuming GC is performed). Each entry in the cache is "touched" whenever it is accessed.

The age is defined in milliseconds.

Default is `10 timutes`.

`mibs_cache_gclimit() = integer() > 0 | infinity <optional>`

When performing a GC, this is the max number of cache entries that will be deleted from the cache.

The reason for having this limit is that if the cache is large, the GC can potentially take a long time, during which the agent is locked.

Default is `100`.

`error_report_mod() = atom() <optional>`

Defines an error report module, implementing the *snmpa_error_report* behaviour. Two modules are provided with the toolkit: `snmpa_error_logger` and `snmpa_error_io`.

Default is `snmpa_error_logger`.

`symbolic_store() = [symbolic_store_opt()]`

`symbolic_store_opt() = {verbosity, verbosity()}`

Defines options specific for the SNMP agent symbolic store.

For defaults see the options in `symbolic_store_opt()`.

`target_cache() = [target_cache_opt()]`

`target_cache_opt() = {verbosity, verbosity()}`

Defines options specific for the SNMP agent target cache.

For defaults see the options in `target_cache_opt()`.

`agent_config() = [agent_config_opt()] <mandatory>`

`agent_config_opt() = {dir, agent_config_dir()} | {force_load, force_load()} | {verbosity, verbosity()}`

Defines specific config related options for the SNMP agent.

For defaults see the options in `agent_config_opt()`.

`agent_config_dir = dir() <mandatory>`

Defines where the SNMP agent configuration files are stored.

`force_load() = bool() <optional>`

If `true` the configuration files are re-read during start-up, and the contents of the configuration database ignored. Thus, if `true`, changes to the configuration database are lost upon reboot of the agent.

Default is `false`.

Manager specific config options and types:

`server() = [server_opt()] <optional>`

`server_opt() = {timeout, server_timeout()} | {verbosity, verbosity()}`

Specifies the options for the manager server process.

Default is `silence`.

`server_timeout() = integer() <optional>`

Asynchroneous request cleanup time. For every requests, some info is stored internally, in order to be able to deliver the reply (when it arrives) to the proper destination. If the reply arrives, this info will be deleted. But if there is no reply (in time), the info has to be deleted after the *best before* time has been passed. This cleanup will be performed at regular intervals, defined by the `server_timeout()` time. The information will have a *best before* time, defined by the `Expire` time given when calling the request function (see *async_get*, *async_get_next* and *async_set*).

Time in milli-seconds.

Default is `30000`.

`manager_config() = [manager_config_opt()] <mandatory>`

`manager_config_opt() = {dir, manager_config_dir()} | {db_dir, manager_db_dir()} | {db_init_error, db_init_error()} | {repair, manager_repair()} | {auto_save, manager_auto_save()} | {verbosity, verbosity()}`

Defines specific config related options for the SNMP manager.

For defaults see the options in `manager_config_opt()`.

`manager_config_dir = dir() <mandatory>`

Defines where the SNMP manager configuration files are stored.

`manager_db_dir = dir() <mandatory>`

Defines where the SNMP manager store persistent data.

`manager_repair() = false | true | force <optional>`

Defines the repair option for the persistent database (if and how the table is repaired when opened).

Default is `true`.

`manager_auto_save() = integer() | infinity <optional>`

The auto save interval. The table is flushed to disk whenever not accessed for this amount of time.

Default is `5000`.

`manager_irb() = auto | user | {user, integer()} <optional>`

This option defines how the manager will handle the sending of response (acknowledgment) to received inform-requests.

- `auto` - The manager will autonomously send response (acknowledgment> to inform-request messages.
- `{user, integer()}` - The manager will send response (acknowledgment) to inform-request messages when the *handle_inform* function completes. The integer is the time, in milli-seconds, that the manager will consider the stored inform-request info valid.
- `user` - Same as `{user, integer()}`, except that the default time, 15000 milli-seconds, is used.

See *snmpm_network_interface*, *handle_inform* and *definition of the manager net if* for more info.

Default is `auto`.

`manager_mibs() = [string()] <optional>`

Specifies a list of MIBs (including path) and defines which MIBs are initially loaded into the SNMP manager.

Default is `[]`.

`manager_net_if() = [manager_net_if_opt()] <optional>`

`manager_net_if_opt() = {module, manager_net_if_module()} | {verbosity, verbosity()} | {options, manager_net_if_options()}`

Defines options specific for the SNMP manager network interface entity.

For defaults see the options in `manager_net_if_opt()`.

`manager_net_if_options() = [manager_net_if_option()] <optional>`

`manager_net_if_option() = {bind_to, bind_to()} | {sndbuf, sndbuf()} | {recbuf, recbuf()} | {no_reuse, no_reuse()} | {filter, manager_net_if_filter_options()}`

These options are actually specific to the used module. The ones shown here are applicable to the default `manager_net_if_module()`.

For defaults see the options in `manager_net_if_option()`.

`manager_net_if_module() = atom() <optional>`

Module which handles the network interface part for the SNMP manager. Must implement the *snmpm_network_interface* behaviour.

Default is `snmpm_net_if`.

`manager_net_if_filter_options() = [manager_net_if_filter_option()] <optional>`

`manager_net_if_filter_option() = {module, manager_net_if_filter_module()}`

These options are actually specific to the used module. The ones shown here are applicable to the default `manager_net_if_filter_module()`.

For defaults see the options in `manager_net_if_filter_option()`.

`manager_net_if_filter_module() = atom() <optional>`

Module which handles the network interface filter part for the SNMP manager. Must implement the *snmpm_network_interface_filter* behaviour.

Default is `snmpm_net_if_filter`.

`def_user_module() = atom() <optional>`

The module implementing the default user. See the *snmpm_user* behaviour.

Default is `snmpm_user_default`.

`def_user_data() = term() <optional>`

Data for the default user. Passed to the user when calling the callback functions.

Default is `undefined`.

Common config types:

`restart_type() = permanent | transient | temporary`

See *supervisor* documentation for more info.

Default is `permanent` for the agent and `transient` for the manager.

`db_init_error() = terminate | create`

Defines what to do if the agent is unable to open an existing database file. `terminate` means that the agent/manager will terminate and `create` means that the agent/manager will remove the faulty file(s) and create new ones.

Default is `terminate`.

`priority() = atom() <optional>`

Defines the Erlang priority for all SNMP processes.

Default is `normal`.

`versions() = [version()] <optional>`

`version() = v1 | v2 | v3`

Which SNMP versions shall be accepted/used.

Default is `[v1,v2,v3]`.

`verbosity() = silence | info | log | debug | trace <optional>`

Verbosity for a SNMP process. This specifies now much debug info is printed.

Default is `silence`.

`bind_to() = bool() <optional>`

If `true`, net_if binds to the IP address. If `false`, net_if listens on any IP address on the host where it is running.

Default is `false`.

`no_reuse() = bool() <optional>`

If `true`, net_if does not specify that the IP and port address should be reusable. If `false`, the address is set to reusable.

Default is `false`.

`recbuf() = integer() <optional>`

Receive buffer size.

Default value is defined by `gen_udp`.

`sndbuf() = integer() <optional>`

Send buffer size.

Default value is defined by `gen_udp`.

`note_store() = [note_store_opt()] <optional>`

`note_store_opt() = {timeout, note_store_timeout()} | {verbosity, verbosity()}`

Specifies the options for the SNMP note store.

For defaults see the options in `note_store_opt()`.

`note_store_timeout() = integer() <optional>`

Note cleanup time. When storing a note in the note store, each note is given lifetime. Every `timeout` the note_store process performs a GC to remove the expired note's. Time in milli-seconds.

Default is `30000`.

`audit_trail_log() [audit_trail_log_opt()] <optional>`

`audit_trail_log_opt() = {type, atl_type()} | {dir, atl_dir()} | {size, atl_size()} | {repair, atl_repair()} | {seqno, atl_seqno()}`

If present, this option specifies the options for the *audit trail logging*. The `disk_log` module is used to maintain a wrap log. If present, the `dir` and `size` options are mandatory.

If not present, audit trail logging is not used.

`atl_type() = read | write | read_write <optional>`

Specifies what type of an audit trail log should be used. The effect of the type is actually different for the the agent and the manager.

For the agent:

- If `write` is specified, only set requests are logged.
- If `read` is specified, only get requests are logged.
- If `read_write`, all requests are logged.

For the manager:

- If `write` is specified, only sent messages are logged.
- If `read` is specified, only received messages are logged.
- If `read_write`, both outgoing and incoming messages are logged.

Default is `read_write`.

`atl_dir = dir() <mandatory>`

Specifies where the audit trail log should be stored.

If `audit_trail_log` specifies that logging should take place, this parameter *must* be defined.

`atl_size() = {integer(), integer()} <mandatory>`

Specifies the size of the audit trail log. This parameter is sent to `disk_log`.

If `audit_trail_log` specifies that logging should take place, this parameter *must* be defined.

`atl_repair() = true | false | truncate | snmp_repair <optional>`

Specifies if and how the audit trail log shall be repaired when opened. Unless this parameter has the value `snmp_repair` it is sent to `disk_log`. If, on the other hand, the value is `snmp_repair`, snmp attempts to handle certain faults on its own. And even if it cannot repair the file, it does not truncate it directly, but instead *moves it aside* for later off-line analysis.

Default is `true`.

`atl_seqno() = true | false <optional>`

Specifies if the audit trail log entries will be (sequence) numbered or not. The range of the sequence numbers are according to RFC 5424, i.e. 1 through 2147483647.

Default is `false`.

1.5.2 Modifying the Configuration Files

To to start the application (agent and/or manager), the configuration files must be modified and there are two ways of doing this. Either edit the files manually, or run the configuration tool as follows.

If authentication or encryption is used (SNMPv3 only), start the `crypto` application.

```
1> snmp:config().

Simple SNMP configuration tool (version 4.0)
------------------------------------------------
Note: Non-trivial configurations still has to be
      done manually. IP addresses may be entered
      as dront.ericsson.se (UNIX only) or
      123.12.13.23
------------------------------------------------

Configure an agent (y/n)? [y]

Agent system config:
--------------------
1. Agent process priority (low/normal/high) [normal]
2. What SNMP version(s) should be used (1,2,3,1&2,1&2&3,2&3)? [3] 1&2&3
```

```
3. Configuration directory (absolute path)? [/ldisk/snmp] /ldisk/snmp/agent/conf
4. Config verbosity (silence/info/log/debug/trace)? [silence]
5. Database directory (absolute path)? [/ldisk/snmp] /ldisk/snmp/agent/db
6. Mib storage type (ets/dets/mnesia)? [ets]
7. Target cache verbosity (silence/info/log/debug/trace)? [silence]
8. Symbolic store verbosity (silence/info/log/debug/trace)? [silence]
9. Local DB verbosity (silence/info/log/debug/trace)? [silence]
10. Local DB repair (true/false/force)? [true]
11. Local DB auto save (infinity/milli seconds)? [5000]
12. Error report module? [snmpa_error_logger]
13. Agent type (master/sub)? [master]
14. Master-agent verbosity (silence/info/log/debug/trace)? [silence] log
15. Shall the agent re-read the configuration files during startup
    (and ignore the configuration database) (true/false)? [true]
16. Multi threaded agent (true/false)? [false] true
17. Check for duplicate mib entries when installing a mib (true/false)? [false]
18. Check for duplicate trap names when installing a mib (true/false)? [false]
19. Mib server verbosity (silence/info/log/debug/trace)? [silence]
20. Mib server cache (true/false)? [true]
21. Note store verbosity (silence/info/log/debug/trace)? [silence]
22. Note store GC timeout? [30000]
23. Shall the agent use an audit trail log (y/n)? [n] y
23b. Audit trail log type (write/read_write)? [read_write]
23c. Where to store the audit trail log? [/ldisk/snmp] /ldisk/snmp/agent/log
23d. Max number of files? [10]
23e. Max size (in bytes) of each file? [10240]
23f. Audit trail log repair (true/false/truncate)? [true]
24. Which network interface module shall be used? [snmpa_net_if]
25. Network interface verbosity (silence/info/log/debug/trace)? [silence] log
25a. Bind the agent IP address (true/false)? [false]
25b. Shall the agents IP address and port be not reusable (true/false)? [false]
25c. Agent request limit (used for flow control) (infinity/pos integer)? [infinity] 32
25d. Receive buffer size of the agent (in bytes) (default/pos integer)? [default]
25e. Send buffer size of the agent (in bytes) (default/pos integer)? [default]
25f. Do you wish to specify a network interface filter module (or use default) [default]

Agent snmp config:
------------------
1. System name (sysName standard variable) [bmk's agent]
2. Engine ID (snmpEngineID standard variable) [bmk's engine]
3. Max message size? [484]
4. The UDP port the agent listens to. (standard 161) [4000]
5. IP address for the agent (only used as id
   when sending traps) [127.0.0.1]
6. IP address for the manager (only this manager
   will have access to the agent, traps are sent
   to this one) [127.0.0.1]
7. To what UDP port at the manager should traps
   be sent (standard 162)? [5000]
8. Do you want a none- minimum- or semi-secure configuration?
   Note that if you chose v1 or v2, you won't get any security for these
   requests (none, minimum, semi_des, semi_aes) [minimum]
making sure crypto server is started...
8b. Give a password of at least length 8. It is used to generate
    private keys for the configuration:  kalle-anka
9. Current configuration files will now be overwritten. Ok (y/n)? [y]

- - - - - - - - - - - - -
Info: 1. SecurityName "initial" has noAuthNoPriv read access
         and authenticated write access to the "restricted"
         subtree.
      2. SecurityName "all-rights" has noAuthNoPriv read/write
         access to the "internet" subtree.
      3. Standard traps are sent to the manager.
      4. Community "public" is mapped to security name "initial".
      5. Community "all-rights" is mapped to security name "all-rights".
The following agent files were written: agent.conf, community.conf,
standard.conf, target_addr.conf, target_params.conf,
notify.conf, vacm.conf and usm.conf
- - - - - - - - - - - - -

Configure a manager (y/n)? [y]

Manager system config:
----------------------
1. Manager process priority (low/normal/high) [normal]
2. What SNMP version(s) should be used (1,2,3,1&2,1&2&3,2&3)? [3] 1&2&3
3. Configuration directory (absolute path)? [/ldisk/snmp] /ldisk/snmp/manager/conf
4. Config verbosity (silence/info/log/debug/trace)? [silence] log
5. Database directory (absolute path)? [/ldisk/snmp] /ldisk/snmp/manager/db
6. Database repair (true/false/force)? [true]
7. Database auto save (infinity/milli seconds)? [5000]
8. Inform request behaviour (auto/user)? [auto]
9. Server verbosity (silence/info/log/debug/trace)? [silence] log
10. Server GC timeout? [30000]
11. Note store verbosity (silence/info/log/debug/trace)? [silence]
12. Note store GC timeout? [30000]
13. Which network interface module shall be used? [snmpm_net_if]
14. Network interface verbosity (silence/info/log/debug/trace)? [silence] log
15. Bind the manager IP address (true/false)? [false]
16. Shall the manager IP address and port be not reusable (true/false)? [false]
17. Receive buffer size of the manager (in bytes) (default/pos integer)? [default]
18. Send buffer size of the manager (in bytes) (default/pos integer)? [default]
19. Shall the manager use an audit trail log (y/n)? [n] y
19b. Where to store the audit trail log? [/ldisk/snmp] /ldisk/snmp/manager/log
19c. Max number of files? [10]
19d. Max size (in bytes) of each file? [10240]
19e. Audit trail log repair (true/false/truncate)? [true]
20. Do you wish to assign a default user [yes] or use
    the default settings [no] (y/n)? [n]

Manager snmp config:
--------------------
1. Engine ID (snmpEngineID standard variable) [bmk's engine]
2. Max message size? [484]
3. IP address for the manager (only used as id
   when sending requests) [127.0.0.1]
4. Port number (standard 162)? [5000]
5. Configure a user of this manager (y/n)? [y]
5b. User id? kalle
5c. User callback module? snmpm_user_default
5d. User (callback) data? [undefined]
5. Configure a user of this manager (y/n)? [y] n
6. Configure an agent handled by this manager (y/n)? [y]
6b. User id? kalle
6c. Target name? [bmk's agent]
6d. Version (1/2/3)? [1] 3
6e. Community string ? [public]
6f. Engine ID (snmpEngineID standard variable) [bmk's engine]
6g. IP address for the agent [127.0.0.1]
6h. The UDP port the agent listens to. (standard 161) [4000]
6i. Retransmission timeout (infinity/pos integer)? [infinity]
6j. Max message size? [484]
6k. Security model (any/v1/v2c/usm)? [any] usm
6l. Security name? ["initial"]
6m. Security level (noAuthNoPriv/authNoPriv/authPriv)? [noAuthNoPriv] authPriv
6. Configure an agent handled by this manager (y/n)? [y] n
7. Configure an usm user handled by this manager (y/n)? [y]
7a. Engine ID [bmk's engine]
7b. User name? hobbes
```

```
7c. Security name? [hobbes]
7d. Authentication protocol (no/sha/md5)? [no] sha
7e  Authentication [sha] key (length 0 or 20)? [""] [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
7d. Priv protocol (no/des/aes)? [no] des
7f  Priv [des] key (length 0 or 16)? [""] 10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25
7. Configure an usm user handled by this manager (y/n)? [y] n
8. Current configuration files will now be overwritten. Ok (y/n)? [y]

- - - - - - - - - - - - -
The following manager files were written: manager.conf, agents.conf , users.conf and usm.conf
- - - - - - - - - - - - -

--------------------
Configuration directory for system file (absolute path)? [/ldisk/snmp]
ok
```

1.5.3 Starting the application

Start Erlang with the command:

```
erl -config /tmp/snmp/sys
```

If authentication or encryption is used (SNMPv3 only), start the `crypto` application. If this step is forgotten, the agent will not start, but report a `{config_error,{unsupported_crypto,_}}` error.

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

```
2> application:start(snmp).
ok
```

1.5.4 Debugging the application

It is possible to debug every (non-supervisor) process of the application (both agent and manager), possibly with the exception of the net_if module(s), which could be supplied by a user of the application). This is done by calling the `snmpa:verbosity/2` and `snmpm:verbosity/2` function(s) and/or using *configuration parameters*. The verbosity itself has several *levels*: `silence | info | log | debug | trace`. For the lowest verbosity `silence`, nothing is printed. The higher the verbosity, the more is printed. Default value is always `silence`.

```
3> snmpa:verbosity(master_agent, log).
ok
5> snmpa:verbosity(net_if, log).
ok
6>
%% Example of output from the agent when a get-next-request arrives:
** SNMP NET-IF LOG:
   got packet from {147,12,12,12}:5000

** SNMP NET-IF MPD LOG:
   v1, community: all-rights

** SNMP NET-IF LOG:
   got pdu from {147,12,12,12}:5000 {pdu, 'get-next-request',
                                          62612569,noError,0,
                                          [{varbind,[1,1],'NULL','NULL',1}]}

** SNMP MASTER-AGENT LOG:
   apply: snmp_generic,variable_func,[get,{sysDescr,persistent}]

** SNMP MASTER-AGENT LOG:
   returned: {value,"Erlang SNMP agent"}

** SNMP NET-IF LOG:
   reply pdu: {pdu,'get-response',62612569,noError,0,
                   [{varbind,[1,3,6,1,2,1,1,1,0],
                             'OCTET STRING',
                             "Erlang SNMP agent",1}]}

** SNMP NET-IF INFO: time in agent: 19711 mysec
```

Other useful function(s) for debugging the agent are:

`snmpa:info/0,1`

info is used to retrieve a list of miscellaneous agent information.

`snmpa:which_aliasnames/0`

which_aliasnames is used to retrieve a list of all alias-names known to the agent.

`snmpa:which_tables/0`

which_tables is used to retrieve a list of all (MIB) tables known to the agent.

`snmpa:which_variables/0`

which_variables is used to retrieve a list of all (MIB) variables known to the agent.

`snmpa:which_notifications/0`

which_notifications is used to retrieve a list of all (MIB) notifications/traps known to the agent.

`snmpa:restart_worker/0,1`

restart_worker is used to restart the worker process of a multi-threaded agent.

`snmpa:restart_set_worker/0,1`

restart_set_worker is used to restart the set-worker process of a multi-threaded agent.

`snmpa_local_db:print/0,1,2`

For example, this function can show the counters `snmpInPkts` and `snmpOutPkts`.

Another usefull way to debug the agent is to pretty-print the content of all the tables and/or variables handled directly by the agent. This can be done by simply calling:

```
snmpa:print_mib_info()
```

See *print_mib_info/0, print_mib_tables/0* or *print_mib_variables/0* for more info.

1.6 Definition of Agent Configuration Files

All configuration data must be included in configuration files that are located in the configuration directory. The name of this directory is given in the `config_dir` configuration parameter. These files are read at start-up, and are used to initialize the SNMPv2-MIB or STANDARD-MIB, SNMP-FRAMEWORK-MIB, SNMP-MPD-MIB, SNMP-VIEW-BASED-ACM-MIB, SNMP-COMMUNITY-MIB, SNMP-USER-BASED-SM-MIB, SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB (refer to the *Management of the Agent* for a description of the MIBs).

The files are:

- `agent.conf`: see *Agent Information*
- `standard.conf`: see *System Information*
- `context.conf`: see *Contexts*
- `community.conf`: see *Communities*
- `target_addr.conf`: see *Target Address Definitions*
- `target_params.conf`: see *Target Parameters Definitions*
- `vacm.conf`: see *MIB Views for VACM*
- `usm.conf`: see *Security data for USM*
- `notify.conf`: see *Notify Definitions*

The directory where the configuration files are found is given as a parameter to the agent.

The entry format in all files are Erlang terms, separated by a '.' and a *newline*. In the following sections, the formats of these terms are described. Comments may be specified as ordinary Erlang comments.

Syntax errors in these files are discovered and reported with the function `config_err/2` of the error report module at start-up.

1.6.1 Agent Information

The agent information should be stored in a file called `agent.conf`.

Each entry is a tuple of size two:

`{AgentVariable, Value}.`

- `AgentVariable` is one of the variables is SNMP-FRAMEWORK-MIB or one of the internal variables `intAgentUDPPort`, which defines which UDP port the agent listens to, or `intAgentIpAddress`, which defines the IP address of the agent.
- `Value` is the value for the variable.

The following example shows a `agent.conf` file:

```
{intAgentUDPPort, 4000}.
{intAgentIpAddress,[141,213,11,24]}.
{snmpEngineID, "mbj's engine"}.
{snmpEngineMaxPacketSize, 484}.
```

The value of `snmpEngineID` is a string, which for a deployed agent should have a very specific structure. See RFC 2271/2571 for details.

1.6.2 Contexts

The context information should be stored in a file called `context.conf`. The default context `""` need not be present.

Each row defines a context in the agent. This information is used in the table `vacmContextTable` in the SNMP-VIEW-BASED-ACM-MIB.

Each entry is a term:

`ContextName.`

- `ContextName` is a string.

1.6.3 System Information

The system information should be stored in a file called `standard.conf`.

Each entry is a tuple of size two:

`{SystemVariable, Value}.`

- `SystemVariable` is one of the variables in the system group, or `snmpEnableAuthenTraps`.
- `Value` is the value for the variable.

The following example shows a valid `standard.conf` file:

```
{sysDescr, "Erlang SNMP agent"}.
{sysObjectID, [1,2,3]}.
{sysContact, "(mbj,eklas)@erlang.ericsson.se"}.
{sysName, "test"}.
{sysServices, 72}.
{snmpEnableAuthenTraps, enabled}.
```

A value must be provided for all variables, which lack default values in the MIB.

1.6.4 Communities

The community information should be stored in a file called `community.conf`. It must be present if the agent is configured for SNMPv1 or SNMPv2c.

An SNMP *community* is a relationship between an SNMP agent and a set of SNMP managers that defines authentication, access control and proxy characteristics.

The corresponding table is `snmpCommunityTable` in the SNMP-COMMUNITY-MIB.

Each entry is a term:

`{CommunityIndex, CommunityName, SecurityName, ContextName, TransportTag}.`

- `CommunityIndex` is a non-empty string.
- `CommunityName` is a string.
- `SecurityName` is a string.
- `ContextName` is a string.
- `TransportTag` is a string.

1.6.5 MIB Views for VACM

The information about MIB Views for VACM should be stored in a file called `vacm.conf`.

The corresponding tables are `vacmSecurityToGroupTable`, `vacmAccessTable` and `vacmViewTreeFamilyTable` in the SNMP-VIEW-BASED-ACM-MIB.

Each entry is one of the terms, one entry corresponds to one row in one of the tables.

`{vacmSecurityToGroup, SecModel, SecName, GroupName}.`

`{vacmAccess, GroupName, Prefix, SecModel, SecLevel, Match, ReadView, WriteView, NotifyView}.`

`{vacmViewTreeFamily, ViewIndex, ViewSubtree, ViewStatus, ViewMask}.`

- `SecModel` is `any`, `v1`, `v2c`, or `usm`.
- `SecName` is a string.
- `GroupName` is a string.
- `Prefix` is a string.
- `SecLevel` is `noAuthNoPriv`, `authNoPriv`, or `authPriv`
- `Match` is `prefix` or `exact`.
- `ReadView` is a string.
- `WriteView` is a string.
- `NotifyView` is a string.
- `ViewIndex` is an integer.
- `ViewSubtree` is a list of integer.
- `ViewStatus` is either `included` or `excluded`
- `ViewMask` is either `null` or a list of ones and zeros. Ones nominate that an exact match is used for this sub-identifier. Zeros are wild-cards which match any sub-identifier. If the mask is shorter than the sub-tree, the tail is regarded as all ones. `null` is shorthand for a mask with all ones.

1.6.6 Security data for USM

The information about Security data for USM should be stored in a file called `usm.conf`, which must be present if the agent is configured for SNMPv3.

The corresponding table is `usmUserTable` in the SNMP-USER-BASED-SM-MIB.

Each entry is a term:

`{EngineID, UserName, SecName, Clone, AuthP, AuthKeyC, OwnAuthKeyC, PrivP, PrivKeyC, OwnPrivKeyC, Public, AuthKey, PrivKey}.`

- `EngineID` is a string.
- `UserName` is a string.
- `SecName` is a string.
- `Clone` is `zeroDotZero` or a list of integers.
- `AuthP` is a `usmNoAuthProtocol`, `usmHMACMD5AuthProtocol`, or `usmHMACSHAAuthProtocol`.
- `AuthKeyC` is a string.
- `OwnAuthKeyC` is a string.
- `PrivP` is a `usmNoPrivProtocol`, `usmDESPrivProtocol` or `usmAesCfb128Protocol`.
- `PrivKeyC` is a string.
- `OwnPrivKeyC` is a string.
- `Public` is a string.
- `AuthKey` is a list (of integer). This is the User's secret localized authentication key. It is not visible in the MIB. The length of this key needs to be 16 if `usmHMACMD5AuthProtocol` is used, and 20 if `usmHMACSHAAuthProtocol` is used.
- `PrivKey` is a list (of integer). This is the User's secret localized encryption key. It is not visible in the MIB. The length of this key needs to be 16 if `usmDESPrivProtocol` or `usmAesCfb128Protocol` is used.

1.6.7 Notify Definitions

The information about Notify Definitions should be stored in a file called `notify.conf`.

The corresponding table is `snmpNotifyTable` in the SNMP-NOTIFICATION-MIB.

Each entry is a term:

`{NotifyName, Tag, Type}.`

- `NotifyName` is a unique non-empty string.
- `Tag` is a string.
- `Type` is `trap` or `inform`.

1.6.8 Target Address Definitions

The information about Target Address Definitions should be stored in a file called `target_addr.conf`.

The corresponding tables are `snmpTargetAddrTable` in the SNMP-TARGET-MIB and `snmpTargetAddrExtTable` in the SNMP-COMMUNITY-MIB.

Each entry is a term:

`{TargetName, Ip, Udp, Timeout, RetryCount, TagList, ParamsName, EngineId}.` or `{TargetName, Ip, Udp, Timeout, RetryCount, TagList, ParamsName, EngineId, TMask, MaxMessageSize}.`

- `TargetName` is a unique non-empty string.
- `Ip` is a list of four integers.
- `Udp` is an integer.
- `Timeout` is an integer.
- `RetryCount` is an integer.
- `TagList` is a string.
- `ParamsName` is a string.
- `EngineId` is a string or the atom `discovery`.
- `TMask` is a string of size 0, or size 6 (default: []).
- `MaxMessageSize` is an integer (default: 2048).

Note that if `EngineId` has the value `discovery`, the agent cannot send `inform` messages to that manager until it has performed the *discovery* process with that manager.

1.6.9 Target Parameters Definitions

The information about Target Parameters Definitions should be stored in a file called `target_params.conf`.

The corresponding table is `snmpTargetParamsTable` in the SNMP-TARGET-MIB.

Each entry is a term:

`{ParamsName, MPModel, SecurityModel, SecurityName, SecurityLevel}.`

- `ParamsName` is a unique non-empty string.
- `MPModel` is `v1`, `v2c` or `v3`
- `SecurityModel` is `v1`, `v2c`, or `usm`.
- `SecurityName` is a string.
- `SecurityLevel` is `noAuthNoPriv`, `authNoPriv` or `authPriv`.

1.7 Definition of Manager Configuration Files

Configuration data may be included in configuration files that is located in the configuration directory. The name of this directory is given in the `config_dir` configuration parameter. These files are read at start-up.

The directory where the configuration files are found is given as a parameter to the manager.

The entry format in all files are Erlang terms, separated by a '.' and a *newline*. In the following sections, the formats of these terms are described. Comments may be specified as ordinary Erlang comments.

If syntax errors are discovered in these files they are reported with the function `config_err/2` of the *error report module* at start-up.

1.7.1 Manager Information

The manager information should be stored in a file called `manager.conf`.

Each entry is a tuple of size two:

`{Variable, Value}.`

- `Variable` is one of the following:
  - `address` - which defines the IP address of the manager. Default is local host.
  - `port` - which defines which UDP port the manager uses for communicating with agents. *Mandatory*.
  - `engine_id` - The `SnmpEngineID` as defined in SNMP-FRAMEWORK-MIB. *Mandatory*.
  - `max_message_size` - The `snmpEngineMaxMessageSize` as defined in SNMP-FRAMEWORK-MIB. *Mandatory*.
- `Value` is the value for the variable.

The following example shows a `manager.conf` file:

```
{address,          [141,213,11,24]}.
{port,             5000}.
{engine_id,        "mgrEngine"}.
{max_message_size, 484}.
```

The value of `engine_id` is a string, which should have a very specific structure. See RFC 2271/2571 for details.

1.7.2 Users

For each *manager user*, the manager needs some information. This information is either added in the `users.conf` config file or by calling the *register_user* function in run-time.

Each row defines a *manager user* of the manager.

Each entry is a tuple of size four:

`{UserId, UserMod, UserData, DefaultAgentConfig}.`

- `UserId` is any term (used to uniquely identify the user).
- `UserMod` is the user callback module (atom).
- `UserData` is any term (passed on to the user when calling the `UserMod`.
- `DefaultAgentConfig` is a list of default agent config's. These values are used as default values when this user registers agents.

1.7.3 Agents

The information needed to handle agents should be stored in a file called `agents.conf`. It is also possible to add agents in run-time by calling the *register_agent*.

Each entry is a tuple:

```
{UserId, TargetName, Comm, Ip, Port, EngineID, Timeout, MaxMessageSize, Version,
SecModel, SecName, SecLevel}.
```

- `UserId` is the identity of the *manager user* responsible for this agent (term).
- `TargetName` is a *unique non-empty* string.
- `Comm` is the community string (string).
- `Ip` is the ip address of the agent (a list of four integers).
- `Port` is the port number of the agent (integer).
- `EngineID` is the engine-id of the agent (string).
- `Timeout` is re-transmission timeout (`infinity` | integer).
- `MaxMessageSize` is the max message size for outgoing messages to this agent (integer).
- `Version` is the version (v1 | v2 | v3).
- `SecModel` is the security model (any | v1 | v2c | usm).
- `SecName` is the security name (string).
- `SecLevel` is security level (noAuthNoPriv | authNoPriv | authPriv).

1.7.4 Security data for USM

The information about Security data for USM should be stored in a file called `usm.conf`, which must be present if the manager wishes to use SNMPv3 when communicating with agents. It is also possible to add usm data in run-time by calling the *register_usm_user*.

The corresponding table is `usmUserTable` in the SNMP-USER-BASED-SM-MIB.

Each entry is a term:

```
{EngineID, UserName, AuthP, AuthKey, PrivP, PrivKey}.
{EngineID, UserName, SecName, AuthP, AuthKey, PrivP, PrivKey}.
```

The first case is when we have the identity-function (`SecName` = `UserName`).

- `EngineID` is a string.
- `UserName` is a string.
- `SecName` is a string.
- `AuthP` is a `usmNoAuthProtocol`, `usmHMACMD5AuthProtocol` or `usmHMACSHAAuthProtocol`.
- `AuthKey` is a list (of integer). This is the User's secret localized authentication key. It is not visible in the MIB. The length of this key needs to be 16 if `usmHMACMD5AuthProtocol` is used and 20 if `usmHMACSHAAuthProtocol` is used.
- `PrivP` is a `usmNoPrivProtocol`, `usmDESPrivProtocol` or `usmAesCfb128Protocol`.
- `PrivKey` is a list (of integer). This is the User's secret localized encryption key. It is not visible in the MIB. The length of this key needs to be 16 if `usmDESPrivProtocol` or `usmAesCfb128Protocol` is used.

1.8 Agent Implementation Example

This *Implementation Example* section describes how an MIB can be implemented with the SNMP Development Toolkit.

The example shown can be found in the toolkit distribution.

The agent is configured with the configuration tool, using default suggestions for everything but the manager node.

1.8.1 MIB

The MIB used in this example is called EX1-MIB. It contains two objects, a variable with a name and a table with friends.

```
EX1-MIB DEFINITIONS ::= BEGIN

          IMPORTS
                  RowStatus      FROM STANDARD-MIB
                  DisplayString  FROM RFC1213-MIB
                  OBJECT-TYPE    FROM RFC-1212
                  ;

          example1       OBJECT IDENTIFIER ::= { experimental 7 }

          myName OBJECT-TYPE
              SYNTAX  DisplayString (SIZE (0..255))
              ACCESS  read-write
              STATUS  mandatory
              DESCRIPTION
                      "My own name"
              ::= { example1 1 }

          friendsTable OBJECT-TYPE
              SYNTAX  SEQUENCE OF FriendsEntry
              ACCESS  not-accessible
              STATUS  mandatory
              DESCRIPTION
                      "A list of friends."
              ::= { example1 4 }

          friendsEntry OBJECT-TYPE
              SYNTAX  FriendsEntry
              ACCESS  not-accessible
              STATUS  mandatory
              DESCRIPTION
                      ""
              INDEX   { fIndex }
              ::= { friendsTable 1 }

          FriendsEntry ::=
              SEQUENCE {
            fIndex
                      INTEGER,
                   fName
                      DisplayString,
                   fAddress
                      DisplayString,
                   fStatus
                      RowStatus              }

          fIndex OBJECT-TYPE
              SYNTAX  INTEGER
              ACCESS  not-accessible
              STATUS  mandatory
               DESCRIPTION
                      "number of friend"
              ::= { friendsEntry 1 }
          fName OBJECT-TYPE
              SYNTAX  DisplayString (SIZE (0..255))
              ACCESS  read-write
              STATUS  mandatory
              DESCRIPTION
                      "Name of friend"
              ::= { friendsEntry 2 }
          fAddress OBJECT-TYPE
              SYNTAX  DisplayString (SIZE (0..255))
              ACCESS  read-write
              STATUS  mandatory
              DESCRIPTION
                      "Address of friend"
              ::= { friendsEntry 3 }
           fStatus OBJECT-TYPE
              SYNTAX      RowStatus
              ACCESS      read-write
              STATUS      mandatory
              DESCRIPTION
                      "The status of this conceptual row."
              ::= { friendsEntry 4 }
          fTrap TRAP-TYPE
              ENTERPRISE  example1
              VARIABLES   { myName, fIndex }
              DESCRIPTION
                          "This trap is sent when something happens to
         the friend specified by fIndex."
              ::= 1
END
```

1.8.2 Default Implementation

Without writing any instrumentation functions, we can compile the MIB and use the default implementation of it. Recall that MIBs imported by "EX1-MIB.mib" must be present and compiled in the current directory ("./STANDARD-MIB.bin","./RFC1213-MIB.bin") when compiling.

```
unix> erl -config ./sys
1> application:start(snmp).
ok
2> snmpc:compile("EX1-MIB").
No accessfunction for 'friendsTable', using default.
No accessfunction for 'myName', using default.
{ok, "EX1-MIB.bin"}
3> snmpa:load_mibs(snmp_master_agent, ["EX1-MIB"]).
ok
```

This MIB is now loaded into the agent, and a manager can ask questions. As an example of this, we start another Erlang system and the simple Erlang manager in the toolkit:

```
1> snmp_test_mgr:start_link([{agent,"dront.ericsson.se"},{community,"all-rights"},
 %% making it understand symbolic names: {mibs,["EX1-MIB","STANDARD-MIB"]}]).
{ok, <0.89.0>}
%% a get-next request with one OID.
2> snmp_test_mgr:gn([[1,3,6,1,3,7]]).
ok
* Got PDU:
[myName,0] = []
```

```
%% A set-request (now using symbolic names for convenience)
3> snmp_test_mgr:s([{[myName,0], "Martin"}]).
ok
* Got PDU:
[myName,0] = "Martin"
%% Try the same get-next request again
4> snmp_test_mgr:gn([[1,3,6,1,3,7]]).
ok
* Got PDU:
[myName,0] = "Martin"
%% ... and we got the new value.
%% you can event do row operations. How to add a row:
5> snmp_test_mgr:s([{[fName,0], "Martin"}, {[fAddress,0],"home"}, {[fStatus,0],4}]).
 %% createAndGo
ok
* Got PDU:
[fName,0] = "Martin"
[fAddress,0] = "home"
[fStatus,0] = 4
6> snmp_test_mgr:gn([[myName,0]]).
ok
* Got PDU:
[fName,0] = "Martin"
7> snmp_test_mgr:gn().
ok
* Got PDU:
[fAddress,0] = "home"
8> snmp_test_mgr:gn().
ok
* Got PDU:
[fStatus,0] = 1
9>
```

1.8.3 Manual Implementation

The following example shows a "manual" implementation of the EX1-MIB in Erlang. In this example, the values of the objects are stored in an Erlang server. The server has a 2-tuple as loop data, where the first element is the value of variable `myName`, and the second is a sorted list of rows in the table `friendsTable`. Each row is a 4-tuple.

> **Note:**
>
> There are more efficient ways to create tables manually, i.e. to use the module `snmp_index`.

Code

```
-module(ex1).
-author('dummy@flop.org').
%% External exports
-export([start/0, my_name/1, my_name/2, friends_table/3]).
%% Internal exports
-export([init/0]).
-define(status_col, 4).
-define(active, 1).
-define(notInService, 2).
-define(notReady, 3).
-define(createAndGo, 4).   % Action; written, not read
-define(createAndWait, 5). % Action; written, not read
```

```
-define(destroy, 6).       % Action; written, not read
start() ->
    spawn(ex1, init, []).
%%----------------------------------------------------------------
%% Instrumentation function for variable myName.
%% Returns: (get) {value, Name}
%%          (set) noError
%%----------------------------------------------------------------
my_name(get) ->
    ex1_server ! {self(), get_my_name},
    Name = wait_answer(),
    {value, Name}.
my_name(set, NewName) ->
    ex1_server ! {self(), {set_my_name, NewName}},
    noError.
%%----------------------------------------------------------------
%% Instrumentation function for table friendsTable.
%%----------------------------------------------------------------
friends_table(get, RowIndex, Cols) ->
    case get_row(RowIndex) of
   {ok, Row} ->
        get_cols(Cols, Row);
   _  ->
        {noValue, noSuchInstance}
    end;
friends_table(get_next, RowIndex, Cols) ->
    case get_next_row(RowIndex) of
   {ok, Row} ->
        get_next_cols(Cols, Row);
   _  ->
       case get_next_row([]) of
     {ok, Row} ->
         % Get next cols from first row.
         NewCols = add_one_to_cols(Cols),
         get_next_cols(NewCols, Row);
     _  ->
        end_of_table(Cols)
        end
    end;
%%----------------------------------------------------------------
%% If RowStatus is set, then:
%%    *) If set to destroy, check that row does exist
%%    *) If set to createAndGo, check that row does not exist AND
%%         that all columns are given values.
%%    *) Otherwise, error (for simplicity).
%% Otherwise, row is modified; check that row exists.
%%----------------------------------------------------------------
friends_table(is_set_ok, RowIndex, Cols) ->
    RowExists =
   case get_row(RowIndex) of
        {ok, _Row} -> true;
       _ -> false
   end,
    case is_row_status_col_changed(Cols) of
   {true, ?destroy} when RowExists == true ->
        {noError, 0};
   {true, ?createAndGo} when RowExists == false,
                                 length(Cols) == 3 ->
        {noError, 0};
   {true, _} ->
       {inconsistentValue, ?status_col};
   false when RowExists == true ->
        {noError, 0};
   _ ->
        [{Col, _NewVal} | _Cols] = Cols,
       {inconsistentName, Col}
      end;
friends_table(set, RowIndex, Cols) ->
    case is_row_status_col_changed(Cols) of
   {true, ?destroy} ->
        ex1_server ! {self(), {delete_row, RowIndex}};
   {true, ?createAndGo} ->
       NewRow = make_row(RowIndex, Cols),
        ex1_server ! {self(), {add_row, NewRow}};
   false ->
       {ok, Row} = get_row(RowIndex),
        NewRow = merge_rows(Row, Cols),
    ex1_server ! {self(), {delete_row, RowIndex}},
       ex1_server ! {self(), {add_row, NewRow}}
   end,
    {noError, 0}.

%%-----------------------------------------------------------------
%% Make a list of {value, Val} of the Row and Cols list.
%%-----------------------------------------------------------------
get_cols([Col | Cols], Row) ->
    [{value, element(Col, Row)} | get_cols(Cols, Row)];
get_cols([], _Row) ->
    [].
%%-----------------------------------------------------------------
%% As get_cols, but the Cols list may contain invalid column
%% numbers. If it does, we must find the next valid column,
%% or return endOfTable.
%%-----------------------------------------------------------------
get_next_cols([Col | Cols], Row) when Col < 2 ->
    [{[2, element(1, Row)], element(2, Row)} |
     get_next_cols(Cols, Row)];
get_next_cols([Col | Cols], Row) when Col > 4 ->
    [endOfTable |
     get_next_cols(Cols, Row)];
get_next_cols([Col | Cols], Row) ->
    [{[Col, element(1, Row)], element(Col, Row)} |
     get_next_cols(Cols, Row)];
get_next_cols([], _Row) ->
    [].
%%-----------------------------------------------------------------
%% Make a list of endOfTable with as many elems as Cols list.
%%-----------------------------------------------------------------
end_of_table([Col | Cols]) ->
    [endOfTable | end_of_table(Cols)];
end_of_table([]) ->
    [].
add_one_to_cols([Col | Cols]) ->
    [Col + 1 | add_one_to_cols(Cols)];
add_one_to_cols([]) ->
    [].
is_row_status_col_changed(Cols) ->
    case lists:keysearch(?status_col, 1, Cols) of
   {value, {?status_col, StatusVal}} ->
        {true, StatusVal};
   _ -> false
    end.
get_row(RowIndex) ->
    ex1_server ! {self(), {get_row, RowIndex}},
    wait_answer().
get_next_row(RowIndex) ->
    ex1_server ! {self(), {get_next_row, RowIndex}},
    wait_answer().
wait_answer() ->
    receive
   {ex1_server, Answer} ->
     Answer
    end.
%%%-------------------------------------------------------------
%%% Server code follows
%%%-------------------------------------------------------------
init() ->
    register(ex1_server, self()),
    loop("", []).

loop(MyName, Table) ->
    receive
   {From, get_my_name} ->
        From ! {ex1_server, MyName},
       loop(MyName, Table);
   {From, {set_my_name, NewName}} ->
        loop(NewName, Table);
   {From, {get_row, RowIndex}} ->
       Res = table_get_row(Table, RowIndex),
       From ! {ex1_server, Res},
       loop(MyName, Table);
   {From, {get_next_row, RowIndex}} ->
       Res = table_get_next_row(Table, RowIndex),
        From ! {ex1_server, Res},
       loop(MyName, Table);
   {From, {delete_row, RowIndex}} ->
    NewTable = table_delete_row(Table, RowIndex),
       loop(MyName, NewTable);
   {From, {add_row, NewRow}} ->
       NewTable = table_add_row(Table, NewRow),
       loop(MyName, NewTable)
    end.
%%%-------------------------------------------------------------
%%% Functions for table operations. The table is represented as
%%% a list of rows.
%%%-------------------------------------------------------------
table_get_row([{Index, Name, Address, Status} | _], [Index]) ->
    {ok, {Index, Name, Address, Status}};
table_get_row([H | T], RowIndex) ->
    table_get_row(T, RowIndex);
table_get_row([], _RowIndex) ->
    no_such_row.
table_get_next_row([Row | T], []) ->
    {ok, Row};
table_get_next_row([Row | T], [Index | _])
when element(1, Row) > Index ->
    {ok, Row};
table_get_next_row([Row | T], RowIndex) ->
    table_get_next_row(T, RowIndex);
table_get_next_row([], RowIndex) ->
    endOfTable.
table_delete_row([{Index, _, _, _} | T], [Index]) ->
    T;
table_delete_row([H | T], RowIndex) ->
    [H | table_delete_row(T, RowIndex)];
table_delete_row([], _RowIndex) ->
    [].
table_add_row([Row | T], NewRow)
  when element(1, Row) > element(1, NewRow) ->
    [NewRow, Row | T];
table_add_row([H | T], NewRow) ->
    [H | table_add_row(T, NewRow)];
table_add_row([], NewRow) ->
    [NewRow].
make_row([Index], [{2, Name}, {3, Address} | _]) ->
```

```
    {Index, Name, Address, ?active}.
merge_rows(Row, [{Col, NewVal} | T]) ->
    merge_rows(setelement(Col, Row, NewVal), T);
merge_rows(Row, []) ->
    Row.
```

Association File

The association file `EX1-MIB.funcs` for the real implementation looks as follows:

```
{myName, {ex1, my_name, []}}.
{friendsTable, {ex1, friends_table, []}}.
```

Transcript

To use the real implementation, we must recompile the MIB and load it into the agent.

```
1> application:start(snmp).
ok
2> snmpc:compile("EX1-MIB").
{ok,"EX1-MIB.bin"}
3> snmpa:load_mibs(snmp_master_agent, ["EX1-MIB"]).
ok
4> ex1:start().
<0.115.0>
%% Now all requests operates on this "real" implementation.
%% The output from the manager requests will *look* exactly the
%% same as for the default implementation.
```

Trap Sending

How to send a trap by sending the `fTrap` from the master agent is shown in this section. The master agent has the MIB `EX1-MIB` loaded, where the trap is defined. This trap specifies that two variables should be sent along with the trap, `myName` and `fIndex`. `fIndex` is a table column, so we must provide its value and the index for the row in the call to `snmpa:send_trap/4`. In the example below, we assume that the row in question is indexed by 2 (the row with `fIndex` 2).

we use a simple Erlang SNMP manager, which can receive traps.

```
[MANAGER]
1> snmp_test_mgr:start_link([{agent,"dront.ericsson.se"},{community,"public"}
 %% does not have write-access
1>{mibs,["EX1-MIB","STANDARD-MIB"]}]).
{ok, <0.100.0>}
2> snmp_test_mgr:s([{[myName,0], "Klas"}]).
ok
* Got PDU:
Received a trap:
      Generic: 4       %% authenticationFailure
   Enterprise: [iso,2,3]
     Specific: 0
   Agent addr: [123,12,12,21]
    TimeStamp: 42993
2>
```

```
[AGENT]
3> snmpa:send_trap(snmp_master_agent, fTrap,"standard trap", [{fIndex,[2],2}]).
[MANAGER]
2>
* Got PDU:
Received a trap:
      Generic: 6
   Enterprise: [example1]
     Specific: 1
   Agent addr: [123,12,12,21]
    TimeStamp: 69649
[myName,0] = "Martin"
[fIndex,2] = 2
2>
```

1.9 Manager Implementation Example

This *Implementation Example* section describes how a simple manager can be implemented with the SNMP Development Toolkit.

The example shown, *ex2*, can be found in the toolkit distribution.

This example has two functions:

- A simple example of how to use the manager component of the SNMP Development Toolkit.
- A simple example of how to write agent test cases, using the new manager.

1.9.1 The example manager

The example manager, `snmp_ex2_manager`, is a simple example of how to implement an snmp manager using the manager component of the SNMP Development Toolkit.

The module exports the following functions:

- start_link/0, start_link/1
- stop/0
- agent/2, agent/3
- sync_get/2, sync_get/3
- sync_get_next/2, sync_get_next/3
- sync_get_bulk/4, sync_get_bulk/5
- sync_set/2, sync_set/3
- oid_to_name/1

This module is also used by the test module described in the next section.

1.9.2 A simple standard test

This simple standard test, `snmp_ex2_simple_standard_test`, a module which, using the `snmp_ex2_manager` described in the previous section, implements a simple agent test utility.

1.10 Instrumentation Functions

A user-defined instrumentation function for each object attaches the managed objects to real resources. This function is called by the agent on a `get` or `set` operation. The function could read some hardware register, perform a calculation, or whatever is necessary to implement the semantics associated with the conceptual variable. These functions must be written both for scalar variables and for tables. They are specified in the association file, which is a text file. In this file, the `OBJECT IDENTIFIER`, or symbolic name for each managed object, is associated with an Erlang tuple `{Module,Function, ListOfExtraArguments}`.

When a managed object is referenced in an SNMP operation, the associated `{Module, Function, ListOfExtraArguments}` is called. The function is applied to some standard arguments (for example, the operation type) and the extra arguments supplied by the user.

Instrumentation functions must be written for `get` and `set` for scalar variables and tables, and for `get-next` for tables only. The `get-bulk` operation is translated into a series of calls to `get-next`.

1.10.1 Instrumentation Functions

The following sections describe how the instrumentation functions should be defined in Erlang for the different operations. In the following, `RowIndex` is a list of key values for the table, and `Column` is a column number.

These functions are described in detail in *Definition of Instrumentation Functions*.

New / Delete Operations

For scalar variables:

```
variable_access(new [, ExtraArg1, ...])
variable_access(delete [, ExtraArg1, ...])
```

For tables:

```
table_access(new [, ExtraArg1, ...])
table_access(delete [, ExtraArg1, ...])
```

These functions are called for each object in an MIB when the MIB is unloaded or loaded, respectively.

Get Operation

For scalar variables:

```
variable_access(get [, ExtraArg1, ...])
```

For tables:

```
table_access(get,RowIndex,Cols [,ExtraArg1, ...])
```

`Cols` is a list of `Column`. The agent will sort incoming variables so that all operations on one row (same index) will be supplied at the same time. The reason for this is that a database normally retrieves information row by row.

These functions must return the current values of the associated variables.

Set Operation

For scalar variables:

```
variable_access(set, NewValue [, ExtraArg1, ...])
```

For tables:

```
table_access(set, RowIndex, Cols [, ExtraArg1,..])
```

`Cols` is a list of tuples `{Column, NewValue}`.

These functions returns `noError` if the assignment was successful, otherwise an error code.

Is-set-ok Operation

As a complement to the `set` operation, it is possible to specify a test function. This function has the same syntax as the set operation above, except that the first argument is `is_set_ok` instead of `set`. This function is called before the variable is set. Its purpose is to ensure that it is permissible to set the variable to the new value.

```
variable_access(is_set_ok, NewValue [, ExtraArg1, ...])
```

For tables:

```
table_access(set, RowIndex, Cols [, ExtraArg1,..])
```

`Cols` is a list of tuples `{Column, NewValue}`.

Undo Operation

A function which has been called with `is_set_ok` will be called again, either with `set` if there was no error, or with `undo`, if an error occurred. In this way, resources can be reserved in the `is_set_ok` operation, released in the `undo` operation, or made permanent in the `set` operation.

```
variable_access(undo, NewValue [, ExtraArg1, ...])
```

For tables:

```
table_access(set, RowIndex, Cols [, ExtraArg1,..])
```

`Cols` is a list of tuples `{Column, NewValue}`.

GetNext Operation

The GetNext Operation operation should only be defined for tables since the agent can find the next instance of plain variables in the MIB and call the instrumentation with the `get` operation.

```
table_access(get_next, RowIndex, Cols [, ExtraArg1, ...])
```

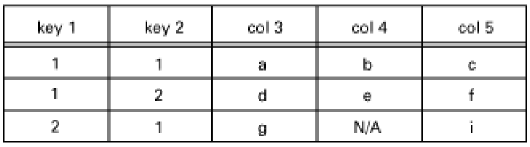
Cols is a list of integers, all greater than or equal to zero. This indicates that the instrumentation should find the next accessible instance. This function returns the tuple {NextOid, NextValue}, or endOfTable. NextOid should be the lexicographically next accessible instance of a managed object in the table. It should be a list of integers, where the first integer is the column, and the rest of the list is the indices for the next row. If endOfTable is returned, the agent continues to search for the next instance among the other variables and tables.

RowIndex may be an empty list, an incompletely specified row index, or the index for an unspecified row.

This operation is best described with an example.

GetNext Example

A table called myTable has five columns. The first two are keys (not accessible), and the table has three rows. The instrumentation function for this table is called my_table.



key 1	key 2	col 3	col 4	col 5
1	1	a	b	c
1	2	d	e	f
2	1	g	N/A	i

Figure 10.1: Contents of my_table

Note:

N/A means not accessible.

The manager issues the following getNext request:

```
getNext{ myTable.myTableEntry.3.1.1,
         myTable.myTableEntry.5.1.1 }
```

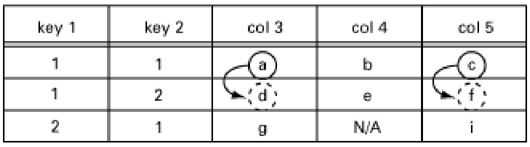
Since both operations involve the 1.1 index, this is transformed into one call to my_table:

```
my_table(get_next, [1, 1], [3, 5])
```

In this call, [1, 1] is the RowIndex, where key 1 has value 1, and key 2 has value 1, and [3, 5] is the list of requested columns. The function should now return the lexicographically next elements:

```
[{[3, 1, 2], d}, {[5, 1, 2], f}]
```

This is illustrated in the following table:



key 1	key 2	col 3	col 4	col 5
1	1	a	b	c
1	2	d	e	f
2	1	g	N/A	i

Figure 10.2: GetNext from [3,1,1] and [5,1,1].

The manager now issues the following `getNext` request:

```
getNext{ myTable.myTableEntry.3.2.1,
         myTable.myTableEntry.5.2.1 }
```

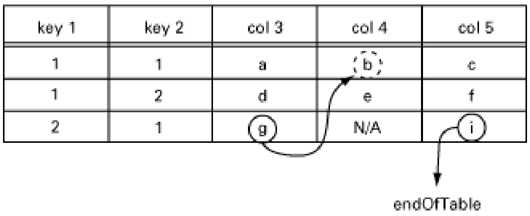
This is transformed into one call to `my_table`:

```
my_table(get_next, [2, 1], [3, 5])
```

The function should now return:

```
[{[4, 1, 1], b}, endOfTable]
```

This is illustrated in the following table:



key 1	key 2	col 3	col 4	col 5
1	1	a	b	c
1	2	d	e	f
2	1	g	N/A	i

Figure 10.3: GetNext from [3,2,1] and [5,2,1].

The manager now issues the following `getNext` request:

```
getNext{ myTable.myTableEntry.3.1.2,
         myTable.myTableEntry.4.1.2 }
```

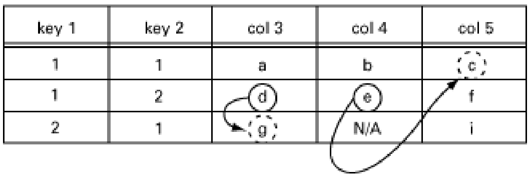
This will be transform into one call to `my_table`:

```
my_table(get_next, [1, 2], [3, 4])
```

The function should now return:

```
[{[3, 2, 1], g}, {[5, 1, 1], c}]
```

This is illustrated in the following table:



key 1	key 2	col 3	col 4	col 5
1	1	a	b	c
1	2	d	e	f
2	1	g	N/A	i

Figure 10.4: GetNext from [3,1,2] and [4,1,2].

The manager now issues the following `getNext` request:

```
getNext{ myTable.myTableEntry,
         myTable.myTableEntry.1.3.2 }
```

This will be transform into two calls to `my_table`:

```
my_table(get_next, [], [0]) and
my_table(get_next, [3, 2], [1])
```

The function should now return:

```
[{[3, 1, 1], a}] and
[{[3, 1, 1], a}]
```

In both cases, the first accessible element in the table should be returned. As the key columns are not accessible, this means that the third column is the first row.

Note:

Normally, the functions described above behave exactly as shown, but they are free to perform other actions. For example, a get-request may have side effects such as setting some other variable, perhaps a global `lastAccessed` variable.

1.10.2 Using the ExtraArgument

The `ListOfExtraArguments` can be used to write generic functions. This list is appended to the standard arguments for each function. Consider two read-only variables for a device, `ipAdr` and `name` with object identifiers 1.1.23.4 and 1.1.7 respectively. To access these variables, one could implement the two Erlang functions `ip_access` and `name_access`, which will be in the MIB. The functions could be specified in a text file as follows:

```
{ipAdr, {my_module, ip_access, []}}.
% Or using the oid syntax for 'name'
{[1,1,7], {my_module, name_access, []}}.
```

The `ExtraArgument` parameter is the empty list. For example, when the agent receives a get-request for the `ipAdr` variable, a call will be made to `ip_access(get)`. The value returned by this function is the answer to the get-request.

If `ip_access` and `name_access` are implemented similarly, we could write a `generic_access` function using the `ListOfExtraArguments`:

```
{ipAdr, {my_module, generic_access, ['IPADR']}}.
% The mnemonic 'name' is more convenient than 1.1.7
{name, {my_module, generic_access, ['NAME']}}.
```

When the agent receives the same get-request as above, a call will be made to `generic_access(get, 'IPADR')`.

Yet another possibility, closer to the hardware, could be:

```
{ipAdr, {my_module, generic_access, [16#2543]}}.
{name, {my_module, generic_access, [16#A2B3]}}.
```

1.10.3 Default Instrumentation

When the MIB definition work is finished, there are two major issues left.

- Implementing the MIB
- Implementing a Manager Application.

Implementing an MIB can be a tedious task. Most probably, there is a need to test the agent before all tables and variables are implemented. In this case, the default instrumentation functions are useful. The toolkit can generate default instrumentation functions for variables as well as for tables. Consequently, a running prototype agent, which can handle `set`, `get`, `get-next` and table operations, is generated without any programming.

The agent stores the values in an internal volatile database, which is based on the standard module `ets`. However, it is possible to let the MIB compiler generate functions which use an internal, persistent database, or the Mnesia DBMS. Refer to the Mnesia User Guide and the Reference Manual, section SNMP, module `snmp_generic` for more information.

When parts of the MIB are implemented, you recompile it and continue on by using default functions. With this approach, the SNMP agent can be developed incrementally.

The default instrumentation allows the application on the manager side to be developed and tested simultaneously with the agent. As soon as the ASN.1 file is completed, let the MIB compiler generate a default implementation and develop the management application from this.

Table Operations

The generation of default functions for tables works for tables which use the `RowStatus` textual convention from SNMPv2, defined in STANDARD-MIB and SNMPv2-TC.

> **Note:**
>
> We strongly encourage the use of the `RowStatus` convention for every table that can be modified from the manager, even for newly designed SNMPv1 MIBs. In SNMPv1, everybody has invented their own scheme for emulating table operations, which has led to numerous inconsistencies. The convention in SNMPv2 is flexible and powerful and has been tested successfully. If the table is read only, no RowStatus column should be used.

1.10.4 Atomic Set

In SNMP, the `set` operation is atomic. Either all variables which are specified in a `set` operation are changed, or none are changed. Therefore, the `set` operation is divided into two phases. In the first phase, the new value of each variable is checked against the definition of the variable in the MIB. The following definitions are checked:

- the type
- the length
- the range
- the variable is writable and within the MIB view.

At the end of phase one, the user defined `is_set_ok` functions are called for each scalar variable, and for each group of table operations.

If no error occurs, the second phase is performed. This phase calls the user defined `set` function for all variables.

If an error occurs, either in the `is_set_ok` phase, or in the `set` phase, all functions which were called with `is_set_ok` but not `set`, are called with `undo`.

There are limitations with this transaction mechanism. If complex dependencies exist between variables, for example between `month` and `day`, another mechanism is needed. Setting the date to 'Feb 31' can be avoided by a somewhat more generic transaction mechanism. You can continue and find more and more complex situations and construct an N-phase set-mechanism. This toolkit only contains a trivial mechanism.

The most common application of transaction mechanisms is to keep row operations together. Since our agent sorts row operations, the mechanism implemented in combination with the RowStatus (particularly 'createAndWait' value) solve most problems elegantly.

1.11 Definition of Instrumentation Functions

The section *Definition of Instrumentation Functions* describes the user defined functions, which the agent calls at different times.

1.11.1 Variable Instrumentation

For scalar variables, a function `f(Operation, ...)` must be defined.

The `Operation` can be `new`, `delete`, `get`, `is_set_ok`, `set`, or `undo`.

In case of an error, all instrumentation functions may return either an SNMPv1 or an SNMPv2 error code. If it returns an SNMPv2 code, it is converted into an SNMPv1 code before it is sent to a SNMPv1 manager. It is recommended to use the SNMPv2 error codes for all instrumentation functions, as these provide more details. See *Appendix A* for a description of error code conversions.

f(new [, ExtraArgs])

The function `f(new [, ExtraArgs])` is called for each variable in the MIB when the MIB is loaded into the agent. This makes it possible to perform necessary initialization.

This function is optional. The return value is discarded.

f(delete [, ExtraArgs])

The function `f(delete [, ExtraArgs])` is called for each object in an MIB when the MIB is unloaded from the agent. This makes it possible to perform necessary clean-up.

This function is optional. The return value is discarded.

f(get [, ExtraArgs])

The function `f(get [, ExtraArgs])` is called when a get-request or a get-next request refers to the variable.

This function is mandatory.

Valid Return Values

- `{value, Value}`. The `Value` must be of correct type, length and within ranges, otherwise `genErr` is returned in the response PDU. If the object is an enumerated integer, the symbolic enum value may be used as an atom. If the object is of type BITS, the return value shall be an integer or a list of bits that are set.
- `{noValue, noSuchName}`(SNMPv1)
- `{noValue, noSuchObject | noSuchInstance}` (SNMPv2)
- `genErr`. Used if an error occurred. Note, this should be an internal processing error, e.g. a caused by a programing fault somewhere. If the variable does not exist, use `{noValue, noSuchName}` or `{noValue, noSuchInstance}`.

f(is_set_ok, NewValue [, ExtraArgs])

The function `f(is_set_ok, NewValue [, ExtraArgs])` is called in phase one of the set-request processing so that the new value can be checked for inconsistencies.

`NewValue` is guaranteed to be of the correct type, length and within ranges, as specified in the MIB. If the object is an enumerated integer or of type BITS, the integer value is used.

This function is optional.

If this function is called, it will be called again, either with `undo` or with `set` as first argument.

Valid return values

- `noError`
- `badValue | noSuchName | genErr`(SNMPv1)
- `noAccess | noCreation | inconsistentValue | resourceUnavailable | inconsistentName | genErr`(SNMPv2)

f(undo, NewValue [, ExtraArgs])

If an error occurred, this function is called after the `is_set_ok` function is called. If `set` is called for this object, `undo` is not called.

`NewValue` is guaranteed to be of the correct type, length and within ranges, as specified in the MIB. If the object is an enumerated integer or of type BITS, the integer value is used.

This function is optional.

Valid return values

- `noError`
- `genErr`(SNMPv1)
- `undoFailed | genErr`(SNMPv2)

f(set, NewValue [, ExtraArgs])

This function is called to perform the set in phase two of the set-request processing. It is only called if the corresponding `is_set_ok` function is present and returns `noError`.

`NewValue` is guaranteed to be of the correct type, length and within ranges, as specified in the MIB. If the object is an enumerated integer or of type BITS, the integer value is used.

This function is mandatory.

Valid return values

- `noError`
- `genErr`(SNMPv1)
- `commitFailed | undoFailed | genErr`(SNMPv2)

1.11.2 Table Instrumentation

For tables, a `f(Operation, ...)` function should be defined (the function shown is exemplified with `f`).

The `Operation` can be `new`, `delete`, `get`, `next`, `is_set_ok`, `undo` or `set`.

In case of an error, all instrumentation functions may return either an SNMPv1 or an SNMPv2 error code. If it returns an SNMPv2 code, it is converted into an SNMPv1 code before it is sent to a SNMPv1 manager. It is recommended to use the SNMPv2 error codes for all instrumentation functions, as these provide more details. See *Appendix A* for a description of error code conversions.

f(new [, ExtraArgs])

The function `f(new [, ExtraArgs])` is called for each object in an MIB when the MIB is loaded into the agent. This makes it possible to perform the necessary initialization.

This function is optional. The return value is discarded.

f(delete [, ExtraArgs])

The function `f(delete [, ExtraArgs])` is called for each object in an MIB when the MIB is unloaded from the agent. This makes it possible to perform any necessary clean-up.

This function is optional. The return value is discarded.

f(get, RowIndex, Cols [, ExtraArgs])

The function `f(get, RowIndex, Cols [, ExtraArgs])` is called when a get-request refers to a table.

This function is mandatory.

Arguments

- `RowIndex` is a list of integers which define the key values for the row. The `RowIndex` is the list representation (list of integers) which follow the `Cols` integer in the OBJECT IDENTIFIER.
- `Cols` is a list of integers which represent the column numbers. The `Cols` are sorted by increasing value and are guaranteed to be valid column numbers.

Valid Return Values

- A list with as many elements as the `Cols` list, where each element is the value of the corresponding column. Each element can be:
  - `{value, Value}`. The `Value` must be of correct type, length and within ranges, otherwise `genErr` is returned in the response PDU. If the object is an enumerated integer, the symbolic enum value may be used (as an atom). If the object is of type BITS, the return value shall be an integer or a list of bits that are set.
  - `{noValue, noSuchName}`(SNMPv1)
  - `{noValue, noSuchObject | noSuchInstance}`(SNMPv2)
- `{noValue, Error}`. If the row does not exist, because all columns have `{noValue, Error}`), the single tuple `{noValue, Error}` can be returned. This is a shorthand for a list with all elements `{noValue, Error}`.
- `genErr`. Used if an error occurred. Note that this should be an internal processing error, e.g. a caused by a programing fault somewhere. If some column does not exist, use `{noValue, noSuchName}` or `{noValue, noSuchInstance}`.

f(get_next, RowIndex, Cols [, ExtraArgs])

The function `f(get_next, RowIndex, Cols [, ExtraArgs])` is called when a get-next- or a get-bulk-request refers to the table.

The `RowIndex` argument may refer to an existing row or a non-existing row, or it may be unspecified. The `Cols` list may refer to inaccessible columns or non-existing columns. For each column in the `Cols` list, the corresponding next instance is determined, and the last part of its OBJECT IDENTIFIER and its value is returned.

This function is mandatory.

Arguments

- `RowIndex` is a list of integers (possibly empty) that defines the key values for a row. The `RowIndex` is the list representation (list of integers), which follow the `Cols` integer in the OBJECT IDENTIFIER.
- `Cols` is a list of integers, greater than or equal to zero, which represents the column numbers.

Valid Return Values

- A list with as many elements as the `Cols` list Each element can be:
  - `{NextOid, NextValue}`, where `NextOid` is the lexicographic next OBJECT IDENTIFIER for the corresponding column. This should be specified as the OBJECT IDENTIFER part following the table entry. This means that the first integer is the column number and the rest is a specification of the keys. `NextValue` is the value of this element.
  - `endOfTable` if there are no accessible elements after this one.
- `{genErr, Column}` where `Column` denotes the column that caused the error. `Column` must be one of the columns in the `Cols` list. Note that this should be an internal processing error, e.g. a caused by a programing fault somewhere. If some column does not exist, you must return the next accessible element (or `endOfTable`).

f(is_set_ok, RowIndex, Cols [, ExtraArgs])

The function `f(is_set_ok, RowIndex, Cols [, ExtraArgs])` is called in phase one of the set-request processing so that new values can be checked for inconsistencies.

If the function is called, it will be called again with `undo`, or with `set` as first argument.

This function is optional.

Arguments

- `RowIndex` is a list of integers which define the key values for the row. The `RowIndex` is the list representation (list of integers) which follow the `Cols` integer in the OBJECT IDENTIFIER.

- `Cols` is a list of `{Column, NewValue}`, where `Column` is an integer, and `NewValue` is guaranteed to be of the correct type, length and within ranges, as specified in the MIB. If the object is an enumerated integer or of type BITS, the integer value is used. The list is sorted by `Column` (increasing) and each `Column` is guaranteed to be a valid column number.

Valid Return Values

- `{noError, 0}`
- `{Error, Column}`, where `Error` is the same as for `is_set_ok` for variables, and `Column` denotes the faulty column. `Column` must be one of the columns in the `Cols` list.

f(undo, RowIndex, Cols [, ExtraArgs])

If an error occurs, The function `f(undo, RowIndex, Cols [, ExtraArgs])` is called after the `is_set_ok` function. If `set` is called for this object, `undo` is not called.

This function is optional.

Arguments

- `RowIndex` is a list of integers which define the key values for the row. The `RowIndex` is the list representation (list of integers) which follow the `Cols` integer in the OBJECT IDENTIFIER.
- `Cols` is a list of `{Column, NewValue}`, where `Column` is an integer, and `NewValue` is guaranteed to be of the correct type, length and within ranges, as specified in the MIB. If the object is an enumerated integer or of type BITS, the integer value is used. The list is sorted by `Column` (increasing) and each `Column` is guaranteed to be a valid column number.

Valid Return Values

- `{noError, 0}`
- `{Error, Column}` where `Error` is the same as for `undo` for variables, and `Column` denotes the faulty column. `Column` must be one of the columns in the `Cols` list.

f(set, RowIndex, Cols [, ExtraArgs])

The function `f(set, RowIndex, Cols [, ExtraArgs])` is called to perform the set in phase two of the set-request processing. It is only called if the corresponding `is_set_ok` function did not exist, or returned `{noError, 0}`.

This function is mandatory.

Arguments

- `RowIndex` is a list of integers that define the key values for the row. The `RowIndex` is the list representation (list of integers) which follow the `Cols` integer in the OBJECT IDENTIFIER.
- `Cols` is a list of `{Column, NewValue}`, where `Column` is an integer, and `NewValue` is guaranteed to be of the correct type, length and within ranges, as specified in the MIB. If the object is an enumerated integer or of type BITS, the integer value is used. The list is sorted by `Column` (increasing) and each `Column` is guaranteed to be a valid column number.

Valid Return Values

- `{noError, 0}`
- `{Error, Column}` where `Error` is the same as `set` for variables, and `Column` denotes the faulty column. `Column` must be one of the columns in the `Cols` list.

1.12 Definition of Agent Net if

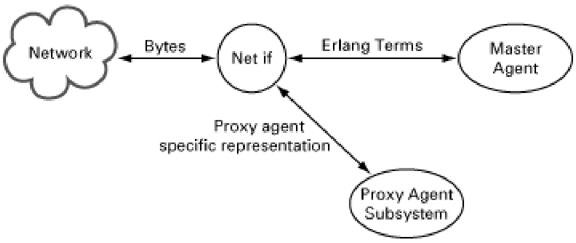


Figure 12.1: The Purpose of Agent Net if

The Network Interface (Net if) process delivers SNMP PDUs to a master agent, and receives SNMP PDUs from the master agent. The most common behaviour of a Net if process is that is receives bytes from a network, decodes them into an SNMP PDU, which it sends to a master agent. When the master agent has processed the PDU, it sends a response PDU to the Net if process, which encodes the PDU into bytes and transmits the bytes onto the network.

However, that simple behaviour can be modified in numerous ways. For example, the Net if process can apply some kind of encrypting/decrypting scheme on the bytes or act as a proxy filter, which sends some packets to a proxy agent and some packets to the master agent.

It is also possible to write your own Net if process. The default Net if process is implemented in the module `snmpa_net_if` and it uses UDP as the transport protocol.

This section describes how to write a Net if process.

1.12.1 Mandatory Functions

A Net if process must implement the SNMP agent *network interface behaviour*.

1.12.2 Messages

The section *Messages* describes mandatory messages, which Net if must send and be able to receive.

Outgoing Messages

Net if must send the following message when it receives an SNMP PDU from the network that is aimed for the MasterAgent:

```
MasterAgent ! {snmp_pdu, Vsn, Pdu, PduMS, ACMData, From, Extra}
```

- `Vsn` is either `'version-1'`, `'version-2'`, or `'version-3'`.
- `Pdu` is an SNMP PDU record, as defined in `snmp_types.hrl`, with the SNMP request.
- `PduMS` is the Maximum Size of the response Pdu allowed. Normally this is returned from `snmpa_mpd:process_packet` (see Reference Manual).
- `ACMData` is data used by the Access Control Module in use. Normally this is returned from `snmpa_mpd:process_packet` (see Reference Manual).
- `From` is the source address. If UDP over IP is used, this should be a 2-tuple `{IP, UDPport}`, where `IP` is a 4-tuple with the IP address, and `UDPport` is an integer.
- `Extra` is any term the Net if process wishes to send to the agent. This term can be retrieved by the instrumentation functions by calling `snmp:current_net_if_data()`. This data is also sent back to the Net if process when the agent generates a response to the request.

The following message is used to report that a response to a request has been received. The only request an agent can send is an Inform-Request.

```
Pid ! {snmp_response_received, Vsn, Pdu, From}
```

- `Pid` is the Process that waits for the response for the request. The Pid was specified in the `send_pdu_req` message *(see below)*.
- `Vsn` is either `'version-1'`, `'version-2'`, or `'version-3'`.
- `Pdu` is the SNMP Pdu received
- `From` is the source address. If UDP over IP is used, this should be a 2-tuple `{IP, UDPport}`, where `IP` is a 4-tuple with the IP address, and `UDPport` is an integer.

Incoming Messages

This section describes the incoming messages which a Net if process must be able to receive.

- `{snmp_response, Vsn, Pdu, Type, ACMData, To, Extra}`

  This message is sent to the Net if process from a master agent as a response to a previously received request.

  - `Vsn` is either `'version-1'`, `'version-2'`, or `'version-3'`.
  - `Pdu` is an SNMP PDU record (as defined in snmp_types.hrl) with the SNMP response.
  - `Type` is the `#pdu.type` of the original request.
  - `ACMData` is data used by the Access Control Module in use. Normally this is just sent to `snmpa_mpd:generate_response_message` (see Reference Manual).
  - `To` is the destination address. If UDP over IP is used, this should be a 2-tuple `{IP, UDPport}`, where `IP` is a 4-tuple with the IP address, and `UDPport` is an integer.
  - `Extra` is the term that the Net if process sent to the agent when the request was sent to the agent.
- `{discarded_pdu, Vsn, ReqId, ACMData, Variable, Extra}`

  This message is sent from a master agent if it for some reason decided to discard the pdu.

  - `Vsn` is either `'version-1'`, `'version-2'`, or `'version-3'`.
  - `ReqId` is the request id of the original request.
  - `ACMData` is data used by the Access Control Module in use. Normally this is just sent to `snmpa_mpd:generate_response_message` (see Reference Manual).
  - `Variable` is the name of an snmp counter that represents the error, e.g. `snmpInBadCommunityUses`.
  - `Extra` is the term that the Net if process sent to the agent when the request was sent to the agent.
- `{send_pdu, Vsn, Pdu, MsgData, To}`

  This message is sent from a master agent when a trap is to be sent.

  - `Vsn` is either `'version-1'`, `'version-2'`, or `'version-3'`.
  - `Pdu` is an SNMP PDU record (as defined in snmp_types.hrl) with the SNMP response.
  - `MsgData` is the message specific data used in the SNMP message. This value is normally sent to `snmpa_mpd:generate_message/4`. In SNMPv1 and SNMPv2c, this message data is the community string. In SNMPv3, it is the context information.
  - `To` is a list of the destination addresses and their corresponding security parameters. This value is normally sent to `snmpa_mpd:generate_message/4`.
- `{send_pdu_req, Vsn, Pdu, MsgData, To, Pid}`

This message is sent from a master agent when a request is to be sent. The only request an agent can send is Inform-Request. The net if process needs to remember the request id and the Pid, and when a response is received for the request id, send it to Pid, using a `snmp_response_received` message.

- `Vsn` is either `'version-1'`, `'version-2'`, or `'version-3'`.
- `Pdu` is an SNMP PDU record (as defined in snmp_types.hrl) with the SNMP response.
- `MsgData` is the message specific data used in the SNMP message. This value is normally sent to `snmpa_mpd:generate_message/4`. In SNMPv1 and SNMPv2c, this message data is the community string. In SNMPv3, it is the context information.
- `To` is a list of the destination addresses and their corresponding security parameters. This value is normally sent to `snmpa_mpd:generate_message/4`.
- `Pid` is a process identifier.

Notes

Since the Net if process is responsible for encoding and decoding of SNMP messages, it must also update the relevant counters in the SNMP group in MIB-II. It can use the functions in the module `snmpa_mpd` for this purpose (refer to the Reference Manual, section `snmp`, module *snmpa_mpd* for more details.)

There are also some useful functions for encoding and decoding of SNMP messages in the module *snmp_pdus*.

1.13 Definition of Manager Net if

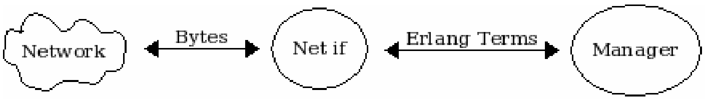


Figure 13.1: The Purpose of Manager Net if

The Network Interface (Net if) process delivers SNMP PDUs to the manager server, and receives SNMP PDUs from the manager server. The most common behaviour of a Net if process is that is receives request PDU from the manager server, encodes the PDU into bytes and transmits the bytes onto the network to an agent. When the reply from the agent is received by the Net if process, which it decodes into an SNMP PDU, which it sends to the manager server.

However, that simple behaviour can be modified in numerous ways. For example, the Net if process can apply some kind of encrypting/decrypting scheme on the bytes.

It is also possible to write your own Net if process. The default Net if process is implemented in the module `snmpm_net_if` and it uses UDP as the transport protocol.

This section describes how to write a Net if process.

1.13.1 Mandatory Functions

A Net if process must implement the SNMP manager *network interface behaviour*.

1.13.2 Messages

The section *Messages* describes mandatory messages, which Net if must send to the manager server process.

Net if must send the following message when it receives an SNMP PDU from the network that is aimed for the MasterAgent:

```
Server ! {snmp_pdu, Pdu, Addr, Port}
```

- `Pdu` is an SNMP PDU record, as defined in `snmp_types.hrl`, with the SNMP request.
- `Addr` is the source address.
- `Port` is port number of the sender.

```
Server ! {snmp_trap, Trap, Addr, Port}
```

- `Trap` is either an SNMP pdu record or an trappdu record, as defined in `snmp_types.hrl`, with the SNMP request.
- `Addr` is the source address.
- `Port` is port number of the sender.

```
Server ! {snmp_inform, Ref, Pdu, PduMS, Addr, Port}
```

- `Ref` is either the atom `ignore` or something that can be used to identify the inform-request (e.g. request-id). `ignore` is used if the response (acknowledgment) to the inform-request has already been sent (this means that the server will not make the call to the *inform_response* function). See the *inform request behaviour* configuration option for more info.
- `Pdu` is an SNMP PDU record, as defined in `snmp_types.hrl`, with the SNMP request.
- `Addr` is the source address.
- `Port` is port number of the sender.

```
Server ! {snmp_report, Data, Addr, Port}
```

- `Data` is either `{ok, Pdu}` or `{error, ReqId, ReasonInfo, Pdu}`. Which one is used depends on the return value from the MPD *process_msg* function. If the MsgData is `ok`, the first is used, and if it is `{error, ReqId, Reason}` the latter is used.
- `Pdu` is an SNMP PDU record, as defined in `snmp_types.hrl`, with the SNMP request.
- `ReqId` is an integer.
- `ReasonInfo` is a term().
- `Addr` is the source address.
- `Port` is port number of the sender.

Notes

Since the Net if process is responsible for encoding and decoding of SNMP messages, it must also update the relevant counters in the SNMP group in MIB-II. It can use the functions in the module `snmpm_mpd` for this purpose (refer to the Reference Manual, section `snmp`, module `snmpm_mpd` for more details).

There are also some useful functions for encoding and decoding of SNMP messages in the module `snmp_pdus`.

1.14 Audit Trail Log

The chapter *Audit Trail Log* describes the audit trail logging.

Both the agent and the manager can be configured to log incoming and outgoing messages. It uses the Erlang standard log mechanism `disk_log` for logging. The size and location of the log files are configurable. A wrap log is used, which means that when the log has grown to a maximum size, it starts from the beginning of the log, overwriting existing log records.

The log can be either a `read`, `write` or a `read_write`.

1.14.1 Agent Logging

For the agent, a `write`, means that all `set` requests and their responses are stored. No `get` requests or traps are stored in a `write`. A `read_write`, all requests, responses and traps are stored.

The log uses a raw data format (basically the BER encoded message), in order to minimize the CPU load needed for the log mechanism. This means that the log is not human readable, but needs to be formatted off-line before it can be read. Use the function *snmpa:log_to_txt* for this purpose.

1.14.2 Manager Logging

For the manager, a `write`, means that all requests (`set` and `get`) and their responses are stored. No traps are stored in a `write`. A `read_write`, all requests, responses and traps are stored.

The log uses a raw data format (basically the BER encoded message), in order to minimize the CPU load needed for the log mechanism. This means that the log is not human readable, but needs to be formatted off-line before it can be read. Use the function *snmpm:log_to_txt* for this purpose.

1.15 Advanced Agent Topics

The chapter *Advanced Agent Topics* describes the more advanced agent related features of the SNMP development tool. The following topics are covered:

- When to use a Sub-agent
- Agent semantics
- Sub-agents and dependencies
- Distributed tables
- Fault tolerance
- Using Mnesia tables as SNMP tables
- Audit Trail Logging
- Deviations from the standard

1.15.1 When to use a Sub-agent

The section *When to use a Sub-agent* describes situations where the mechanism of loading and unloading MIBs is insufficient. In these cases a sub-agent is needed.

Special Set Transaction Mechanism

Each sub-agent can implement its own mechanisms for `set`, `get` and `get-next`. For example, if the application requires the `get` mechanism to be asynchronous, or needs a N-phase `set` mechanism, a specialized sub-agent should be used.

The toolkit allows different kinds of sub-agents at the same time. Accordingly, different MIBs can have different `set` or `get` mechanisms.

Process Communication

A simple distributed agent can be managed without sub-agents. The instrumentation functions can use distributed Erlang to communicate with other parts of the application. However, a sub-agent can be used on each node if this generates too much unnecessary traffic. A sub-agent processes requests per incoming SNMP request, not per variable. Therefore the network traffic is minimized.

If the instrumentation functions communicate with UNIX processes, it might be a good idea to use a special sub-agent. This sub-agent sends the SNMP request to the other process in one packet in order to minimize context switches. For example, if a whole MIB is implemented on the C level in UNIX, but you still want to use the Erlang SNMP tool, then you may have one special sub-agent, which sends the variables in the request as a single operation down to C.

Frequent Loading of MIBs

Loading and unloading of MIBs are quite cheap operations. However, if the application does this very often, perhaps several times per minute, it should load the MIBs once and for all in a sub-agent. This sub-agent only registers and unregisters itself under another agent instead of loading the MIBs each time. This is cheaper than loading an MIB.

Interaction With Other SNMP Agent Toolkits

If the SNMP agent needs to interact with sub-agents constructed in another package, a special sub-agent should be used, which communicates through a protocol specified by the other package.

1.15.2 Agent Semantics

The agent can be configured to be multi-threaded, to process one incoming request at a time, or to have a request limit enabled (this can be used for load control or to limit the effect of DoS attacks). If it is multi-threaded, read requests (`get`, `get-next` and `get-bulk`) and traps are processed in parallel with each other and `set` requests. However, all `set` requests are serialized, which means that if the agent is waiting for the application to complete a complicated write operation, it will not process any new write requests until this operation is finished. It processes read requests and sends traps, concurrently. The reason for not handle write requests in parallel is that a complex locking mechanism would be needed even in the simplest cases. Even with the scheme described above, the user must be careful not to violate that the `set` requests are atoms. If this is hard to do, do not use the multi-threaded feature.

The order within an request is undefined and variables are not processed in a defined order. Do not assume that the first variable in the PDU will be processed before the second, even if the agent processes variables in this order. It cannot even be assumed that requests belonging to different sub-agents have any order.

If the manager tries to set the same variable many times in the same PDU, the agent is free to improvise. There is no definition which determines if the instrumentation will be called once or twice. If called once only, there is no definition that determines which of the new values is going to be supplied.

When the agent receives a request, it keeps the request ID for one second after the response is sent. If the agent receives another request with the same request ID during this time, from the same IP address and UDP port, that request will be discarded. This mechanism has nothing to do with the function `snmpa:current_request_id/0`.

1.15.3 Sub-agents and Dependencies

The toolkit supports the use of different types of sub-agents, but not the construction of sub-agents.

Also, the toolkit does not support dependencies between sub-agents. A sub-agent should by definition be stand alone and it is therefore not good design to create dependencies between them.

1.15.4 Distributed Tables

A common situation in more complex systems is that the data in a table is distributed. Different table rows are implemented in different places. Some SNMP tool-kits dedicate an SNMP sub-agent for each part of the table and load the corresponding MIB into all sub-agents. The Master Agent is responsible for presenting the distributed table as a single table to the manager. The toolkit supplied uses a different method.

The method used to implement distributed tables with this SNMP tool is to implement a table coordinator process responsible for coordinating the processes, which hold the table data and they are called table holders. All table holders must in some way be known by the coordinator; the structure of the table data determines how this is achieved. The coordinator may require that the table holders explicitly register themselves and specify their information. In other cases, the table holders can be determined once at compile time.

When the instrumentation function for the distributed table is called, the request should be forwarded to the table coordinator. The coordinator finds the requested information among the table holders and then returns the answer to the instrumentation function. The SNMP toolkit contains no support for coordination of tables since this must be independent of the implementation.

The advantages of separating the table coordinator from the SNMP tool are:

- We do not need a sub-agent for each table holder. Normally, the sub-agent is needed to take care of communication, but in Distributed Erlang we use ordinary message passing.
- Most likely, some type of table coordinator already exists. This process should take care of the instrumentation for the table.
- The method used to present a distributed table is strongly application dependent. The use of different masking techniques is only valid for a small subset of problems and registering every row in a distributed table makes it non-distributed.

1.15.5 Fault Tolerance

The SNMP agent toolkit gets input from three different sources:

- UDP packets from the network
- return values from the user defined instrumentation functions
- return values from the MIB.

The agent is highly fault tolerant. If the manager gets an unexpected response from the agent, it is possible that some instrumentation function has returned an erroneous value. The agent will not crash even if the instrumentation does. It should be noted that if an instrumentation function enters an infinite loop, the agent will also be blocked forever. The supervisor ,or the application, specifies how to restart the agent.

Using the SNMP Agent in a Distributed Environment

The normal way to use the agent in a distributed environment is to use one master agent located at one node, and zero or more sub-agents located on other nodes. However, this configuration makes the master agent node a single point of failure. If that node goes down, the agent will not work.

One solution to this problem is to make the snmp application a distributed Erlang application, and that means, the agent may be configured to run on one of several nodes. If the node where it runs goes down, another node restarts the agent. This is called *failover*. When the node starts again, it may *takeover* the application. This solution to the problem adds another problem. Generally, the new node has another IP address than the first one, which may cause problems in the communication between the SNMP managers and the agent.

If the snmp agent is configured as a distributed Erlang application, it will during takeover try to load the same MIBs that were loaded at the old node. It uses the same filenames as the old node. If the MIBs are not located in the same paths at the different nodes, the MIBs must be loaded explicitly after takeover.

1.15.6 Using Mnesia Tables as SNMP Tables

The Mnesia DBMS can be used for storing data of SNMP tables. This means that an SNMP table can be implemented as a Mnesia table, and that a Mnesia table can be made visible via SNMP. This mapping is largely automated.

There are three main reasons for using this mapping:

- We get all features of Mnesia, such as fault tolerance, persistent data storage, replication, and so on.
- Much of the work involved is automated. This includes `get-next` processing and `RowStatus` handling.
- The table may be used as an ordinary Mnesia table, using the Mnesia API internally in the application at the same time as it is visible through SNMP.

When this mapping is used, insertion and deletion in the original Mnesia table is slower, with a factor O(log n). The read access is not affected.

A drawback with implementing an SNMP table as a Mnesia table is that the internal resource is forced to use the table definition from the MIB, which means that the external data model must be used internally. Actually, this is only partially true. The Mnesia table may extend the SNMP table, which means that the Mnesia table may have columns which are use internally and are not seen by SNMP. Still, the data model from SNMP must be maintained. Although this is undesirable, it is a pragmatic compromise in many situations where simple and efficient implementation is preferable to abstraction.

Creating the Mnesia Table

The table must be created in Mnesia before the manager can use it. The table must be declared as type `snmp`. This makes the table ordered in accordance with the lexicographical ordering rules of SNMP. The name of the Mnesia table must be identical to the SNMP table name. The types of the INDEX fields in the corresponding SNMP table must be specified.

If the SNMP table has more than one INDEX column, the corresponding Mnesia row is a tuple, where the first element is a tuple with the INDEX columns. Generally, if the SNMP table has *N* INDEX columns and *C* data columns, the Mnesia table is of arity *(C-N)+1*, where the key is a tuple of arity *N* if *N > 1*, or a single term if *N = 1*.

Refer to the Mnesia User's Guide for information on how to declare a Mnesia table as an SNMP table.

The following example illustrates a situation in which we have an SNMP table that we wish to implement as a Mnesia table. The table stores information about employees at a company. Each employee is indexed with the department number and the name.

```
empTable OBJECT-TYPE
       SYNTAX      SEQUENCE OF EmpEntry
       ACCESS      not-accessible
       STATUS      mandatory
       DESCRIPTION
               "A table with information about employees."
::= { emp 1}
empEntry OBJECT-TYPE
       SYNTAX      EmpEntry
       ACCESS      not-accessible
       STATUS      mandatory
       DESCRIPTION
          ""
       INDEX      { empDepNo, empName }
::= { empTable 1 }
EmpEntry ::=
       SEQUENCE {
           empDepNo         INTEGER,
           empName          DisplayString,
           empTelNo         DisplayString
           empStatus        RowStatus
       }
```

The corresponding Mnesia table is specified as follows:

```
mnesia:create_table([{name, employees},
                     {snmp, [{key, {integer, string}}]},
                     {attributes, [key, telno, row_status]}]).
```

Note:

In the Mnesia tables, the two key columns are stored as a tuple with two elements. Therefore, the arity of the table is 3.

Instrumentation Functions

The MIB table shown in the previous section can be compiled as follows:

```
1> snmpc:compile("EmpMIB", [{db, mnesia}]).
```

This is all that has to be done! Now the manager can read, add, and modify rows. Also, you can use the ordinary Mnesia API to access the table from your programs. The only explicit action is to create the Mnesia table, an action the user has to perform in order to create the required table schemas.

Adding Own Actions

It is often necessary to take some specific action when a table is modified. This is accomplished with an instrumentation function. It executes some specific code when the table is set, and passes all other requests down to the pre-defined function.

The following example illustrates this idea:

```
emp_table(set, RowIndex, Cols) ->
    notify_internal_resources(RowIndex, Cols),
    snmp_generic:table_func(set, RowIndex, Cols, {empTable, mnesia});
emp_table(Op, RowIndex, Cols) ->
    snmp_generic:table_func(Op, RowIndex, Cols, {empTable, mnesia}).
```

The default instrumentation functions are defined in the module `snmp_generic`. Refer to the Reference Manual, section SNMP, module `snmp_generic` for details.

Extending the Mnesia Table

A table may contain columns that are used internally, but should not be visible to a manager. These internal columns must be the last columns in the table. The `set` operation will not work with this arrangement, because there are columns that the agent does not know about. This situation is handled by adding values for the internal columns in the `set` function.

To illustrate this, suppose we extend our Mnesia `empTable` with one internal column. We create it as before, but with an arity of 4, by adding another attribute.

```
mnesia:create_table([{name, employees},
                     {snmp, [{key, {integer, string}}]},
                     {attributes, {key, telno, row_status, internal_col}}]).
```

The last column is the internal column. When performing a `set` operation, which creates a row, we must give a value to the internal column. The instrumentation functions will now look as follows:

```
-define(createAndGo, 4).
-define(createAndWait, 5).

emp_table(set, RowIndex, Cols) ->
  notify_internal_resources(RowIndex, Cols),
  NewCols =
    case is_row_created(empTable, Cols) of
      true -> Cols ++ [{4, "internal"}]; % add internal column
      false -> Cols                      % keep original cols
  end,
  snmp_generic:table_func(set, RowIndex, NewCols, {empTable, mnesia});
emp_table(Op, RowIndex, Cols) ->
  snmp_generic:table_func(Op, RowIndex, Cols, {empTable, mnesia}).

is_row_created(Name, Cols) ->
  case snmp_generic:get_status_col(Name, Cols) of
    {ok, ?createAndGo} -> true;
    {ok, ?createAndWait} -> true;
    _ -> false
  end.
```

If a row is created, we always set the internal column to `"internal"`.

1.15.7 Deviations from the Standard

In some aspects the agent does not implement SNMP fully. Here are the differences:

- The default functions and `snmp_generic` cannot handle an object of type `NetworkAddress` as INDEX (SNMPv1 only!). Use `IpAddress` instead.
- The agent does not check complex ranges specified for INTEGER objects. In these cases it just checks that the value lies within the minimum and maximum values specified. For example, if the range is specified as `1..10 | 12..20` the agent would let 11 through, but not 0 or 21. The instrumentation functions must check the complex ranges itself.
- The agent will never generate the `wrongEncoding` error. If a variable binding is erroneous encoded, the `asn1ParseError` counter will be incremented.
- A `tooBig` error in an SNMPv1 packet will always use the `'NULL'` value in all variable bindings.
- The default functions and `snmp_generic` do not check the range of each OCTET in textual conventions derived from OCTET STRING, e.g. `DisplayString` and `DateAndTime`. This must be checked in an overloaded `is_set_ok` function.

1.16 SNMP Appendix A

1.16.1 Appendix A

This appendix describes the conversion of SNMPv2 to SNMPv1 error messages. The instrumentation functions should return v2 error messages.

Mapping of SNMPv2 error message to SNMPv1:

SNMPv2 message	SNMPv1 message
noError	noError
genErr	genErr
noAccess	noSuchName
wrongType	badValue
wrongLength	badValue
wrongEncoding	badValue
wrongValue	badValue
noCreation	noSuchName
inconsistentValue	badValue
resourceUnavailable	genErr
commitFailed	genErr
undoFailed	genErr
notWritable	noSuchName
inconsistentName	noSuchName

Table 16.1: Error Messages

1.17 SNMP Appendix B

1.17.1 Appendix B

RowStatus (from RFC1903)

```
RowStatus ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
            "The RowStatus textual convention is used to manage the
            creation and deletion of conceptual rows, and is used as the
            value of the SYNTAX clause for the status column of a
            conceptual row (as described in Section 7.7.1 in RFC1902.)

            The status column has six defined values:

                 - `active', which indicates that the conceptual row is
                 available for use by the managed device;

                 - `notInService', which indicates that the conceptual
                 row exists in the agent, but is unavailable for use by
                 the managed device (see NOTE below);

                 - `notReady', which indicates that the conceptual row
                 exists in the agent, but is missing information
                 necessary in order to be available for use by the
                 managed device;

                 - `createAndGo', which is supplied by a management
                 station wishing to create a new instance of a
                 conceptual row and to have its status automatically set
                 to active, making it available for use by the managed
                 device;

                 - `createAndWait', which is supplied by a management
                 station wishing to create a new instance of a
                 conceptual row (but not make it available for use by
                 the managed device); and,

                 - `destroy', which is supplied by a management station
                 wishing to delete all of the instances associated with
                 an existing conceptual row.

            Whereas five of the six values (all except `notReady') may
            be specified in a management protocol set operation, only
            three values will be returned in response to a management
            protocol retrieval operation:  `notReady', `notInService' or
            `active'.  That is, when queried, an existing conceptual row
            has only three states:  it is either available for use by
            the managed device (the status column has value `active');
            it is not available for use by the managed device, though
            the agent has sufficient information to make it so (the
            status column has value `notInService'); or, it is not
            available for use by the managed device, and an attempt to
            make it so would fail because the agent has insufficient
            information (the state column has value `notReady').


                                     NOTE WELL

                 This textual convention may be used for a MIB table,
                 irrespective of whether the values of that table's
                 conceptual rows are able to be modified while it is
                 active, or whether its conceptual rows must be taken
                 out of service in order to be modified.  That is, it is
                 the responsibility of the DESCRIPTION clause of the
                 status column to specify whether the status column must
                 not be `active' in order for the value of some other
                 column of the same conceptual row to be modified.  If
                 such a specification is made, affected columns may be
                 changed by an SNMP set PDU if the RowStatus would not
                 be equal to `active' either immediately before or after
                 processing the PDU.  In other words, if the PDU also
                 contained a varbind that would change the RowStatus
                 value, the column in question may be changed if the
                 RowStatus was not equal to `active' as the PDU was
                 received, or if the varbind sets the status to a value
                 other than 'active'.


            Also note that whenever any elements of a row exist, the
            RowStatus column must also exist.

            To summarize the effect of having a conceptual row with a
            status column having a SYNTAX clause value of RowStatus,
            consider the following state diagram:
```

```
                                         STATE
              +--------------+-----------+-------------+-------------
              |      A       |     B     |      C      |      D
              |              |status col.|status column|
              |status column |    is     |      is     |status column
    ACTION    |does not exist|  notReady | notInService|  is active
--------------+--------------+-----------+-------------+-------------
set status    |noError    ->D|inconsist- |inconsistent-|inconsistent-
column to     |       or     |   entValue|        Value|        Value
createAndGo   |inconsistent- |           |             |
              |         Value|           |             |
--------------+--------------+-----------+-------------+-------------
set status    |noError  see 1|inconsist- |inconsistent-|inconsistent-
column to     |       or     |   entValue|        Value|        Value
createAndWait |wrongValue    |           |             |
--------------+--------------+-----------+-------------+-------------
set status    |inconsistent- |inconsist- |noError      |noError
column to     |         Value|   entValue|             |
active        |              |           |             |
              |              |     or    |             |
              |              |           |             |
              |              |see 2   ->D|          ->D|          ->D
--------------+--------------+-----------+-------------+-------------
set status    |inconsistent- |inconsist- |noError      |noError   ->C
column to     |         Value|   entValue|             |
notInService  |              |           |             |
              |              |     or    |             |      or
              |              |           |             |
              |              |see 3   ->C|          ->C|wrongValue
--------------+--------------+-----------+-------------+-------------
set status    |noError       |noError    |noError      |noError
column to     |              |           |             |
destroy       |           ->A|        ->A|          ->A|          ->A
--------------+--------------+-----------+-------------+-------------
set any other |see 4         |noError    |noError      |see 5
column to some|              |           |             |
value         |              |      see 1|          ->C|          ->D
--------------+--------------+-----------+-------------+-------------

            (1) goto B or C, depending on information available to the
            agent.

            (2) if other variable bindings included in the same PDU,
            provide values for all columns which are missing but
            required, then return noError and goto D.

            (3) if other variable bindings included in the same PDU,
            provide values for all columns which are missing but
            required, then return noError and goto C.

            (4) at the discretion of the agent, the return value may be
            either:

                 inconsistentName:  because the agent does not choose to
                 create such an instance when the corresponding
                 RowStatus instance does not exist, or

                 inconsistentValue:  if the supplied value is
                 inconsistent with the state of some other MIB object's
                 value, or

                 noError: because the agent chooses to create the
                 instance.
```

```
If noError is returned, then the instance of the status
column must also be created, and the new state is B or C,
depending on the information available to the agent.  If
inconsistentName or inconsistentValue is returned, the row
remains in state A.

(5) depending on the MIB definition for the column/table,
either noError or inconsistentValue may be returned.

NOTE: Other processing of the set request may result in a
response other than noError being returned, e.g.,
wrongValue, noCreation, etc.


                  Conceptual Row Creation

There are four potential interactions when creating a
conceptual row:  selecting an instance-identifier which is
not in use; creating the conceptual row; initializing any
objects for which the agent does not supply a default; and,
making the conceptual row available for use by the managed
device.


Interaction 1: Selecting an Instance-Identifier

The algorithm used to select an instance-identifier varies
for each conceptual row.  In some cases, the instance-
identifier is semantically significant, e.g., the
destination address of a route, and a management station
selects the instance-identifier according to the semantics.

In other cases, the instance-identifier is used solely to
distinguish conceptual rows, and a management station
without specific knowledge of the conceptual row might
examine the instances present in order to determine an
unused instance-identifier.  (This approach may be used, but
it is often highly sub-optimal; however, it is also a
questionable practice for a naive management station to
attempt conceptual row creation.)

Alternately, the MIB module which defines the conceptual row
might provide one or more objects which provide assistance
in determining an unused instance-identifier.  For example,
if the conceptual row is indexed by an integer-value, then
an object having an integer-valued SYNTAX clause might be
defined for such a purpose, allowing a management station to
issue a management protocol retrieval operation.  In order
to avoid unnecessary collisions between competing management
stations, `adjacent' retrievals of this object should be
different.

Finally, the management station could select a pseudo-random
number to use as the index.  In the event that this index
was already in use and an inconsistentValue was returned in
response to the management protocol set operation, the
management station should simply select a new pseudo-random
number and retry the operation.

A MIB designer should choose between the two latter
algorithms based on the size of the table (and therefore the
efficiency of each algorithm).  For tables in which a large
number of entries are expected, it is recommended that a MIB
object be defined that returns an acceptable index for
```

```
creation.  For tables with small numbers of entries, it is
recommended that the latter pseudo-random index mechanism be
used.


Interaction 2: Creating the Conceptual Row

Once an unused instance-identifier has been selected, the
management station determines if it wishes to create and
activate the conceptual row in one transaction or in a
negotiated set of interactions.

Interaction 2a: Creating and Activating the Conceptual Row

The management station must first determine the column
requirements, i.e., it must determine those columns for
which it must or must not provide values.  Depending on the
complexity of the table and the management station's
knowledge of the agent's capabilities, this determination
can be made locally by the management station.  Alternately,
the management station issues a management protocol get
operation to examine all columns in the conceptual row that
it wishes to create.  In response, for each column, there
are three possible outcomes:

     - a value is returned, indicating that some other
     management station has already created this conceptual
     row.  We return to interaction 1.

     - the exception `noSuchInstance' is returned,
     indicating that the agent implements the object-type
     associated with this column, and that this column in at
     least one conceptual row would be accessible in the MIB
     view used by the retrieval were it to exist. For those
     columns to which the agent provides read-create access,
     the `noSuchInstance' exception tells the management
     station that it should supply a value for this column
     when the conceptual row is to be created.

     - the exception `noSuchObject' is returned, indicating
     that the agent does not implement the object-type
     associated with this column or that there is no
     conceptual row for which this column would be
     accessible in the MIB view used by the retrieval.  As
     such, the management station cannot issue any
     management protocol set operations to create an
     instance of this column.

Once the column requirements have been determined, a
management protocol set operation is accordingly issued.
This operation also sets the new instance of the status
column to `createAndGo'.

When the agent processes the set operation, it verifies that
it has sufficient information to make the conceptual row
available for use by the managed device.  The information
available to the agent is provided by two sources:  the
management protocol set operation which creates the
conceptual row, and, implementation-specific defaults
supplied by the agent (note that an agent must provide
implementation-specific defaults for at least those objects
which it implements as read-only).  If there is sufficient
information available, then the conceptual row is created, a
`noError' response is returned, the status column is set to
`active', and no further interactions are necessary (i.e.,
interactions 3 and 4 are skipped).  If there is insufficient
information, then the conceptual row is not created, and the
set operation fails with an error of `inconsistentValue'.
On this error, the management station can issue a management
protocol retrieval operation to determine if this was
because it failed to specify a value for a required column,
or, because the selected instance of the status column
already existed.  In the latter case, we return to
interaction 1.  In the former case, the management station
can re-issue the set operation with the additional
information, or begin interaction 2 again using
`createAndWait' in order to negotiate creation of the
conceptual row.

                         NOTE WELL

     Regardless of the method used to determine the column
     requirements, it is possible that the management
     station might deem a column necessary when, in fact,
     the agent will not allow that particular columnar
     instance to be created or written.  In this case, the
     management protocol set operation will fail with an
     error such as `noCreation' or `notWritable'.  In this
     case, the management station decides whether it needs
     to be able to set a value for that particular columnar
     instance.  If not, the management station re-issues the
     management protocol set operation, but without setting
     a value for that particular columnar instance;
     otherwise, the management station aborts the row
     creation algorithm.

Interaction 2b: Negotiating the Creation of the Conceptual
Row

The management station issues a management protocol set
operation which sets the desired instance of the status
column to `createAndWait'.  If the agent is unwilling to
process a request of this sort, the set operation fails with
an error of `wrongValue'.  (As a consequence, such an agent
must be prepared to accept a single management protocol set
operation, i.e., interaction 2a above, containing all of the
columns indicated by its column requirements.)  Otherwise,
the conceptual row is created, a `noError' response is
returned, and the status column is immediately set to either
`notInService' or `notReady', depending on whether it has
sufficient information to make the conceptual row available
for use by the managed device.  If there is sufficient
information available, then the status column is set to
`notInService'; otherwise, if there is insufficient
information, then the status column is set to `notReady'.
Regardless, we proceed to interaction 3.

Interaction 3: Initializing non-defaulted Objects

The management station must now determine the column
requirements.  It issues a management protocol get operation
to examine all columns in the created conceptual row.  In
the response, for each column, there are three possible
outcomes:

     - a value is returned, indicating that the agent
     implements the object-type associated with this column
     and had sufficient information to provide a value.  For
     those columns to which the agent provides read-create
     access (and for which the agent allows their values to
     be changed after their creation), a value return tells
     the management station that it may issue additional
     management protocol set operations, if it desires, in
     order to change the value associated with this column.

     - the exception `noSuchInstance' is returned,
     indicating that the agent implements the object-type
     associated with this column, and that this column in at
     least one conceptual row would be accessible in the MIB
     view used by the retrieval were it to exist. However,
     the agent does not have sufficient information to
     provide a value, and until a value is provided, the
     conceptual row may not be made available for use by the
     managed device.  For those columns to which the agent
     provides read-create access, the `noSuchInstance'
     exception tells the management station that it must
     issue additional management protocol set operations, in
     order to provide a value associated with this column.

     - the exception `noSuchObject' is returned, indicating
     that the agent does not implement the object-type
     associated with this column or that there is no
     conceptual row for which this column would be
     accessible in the MIB view used by the retrieval.  As
     such, the management station cannot issue any
     management protocol set operations to create an
     instance of this column.

If the value associated with the status column is
`notReady', then the management station must first deal with
all `noSuchInstance' columns, if any.  Having done so, the
value of the status column becomes `notInService', and we
proceed to interaction 4.

Interaction 4: Making the Conceptual Row Available

Once the management station is satisfied with the values
associated with the columns of the conceptual row, it issues
a management protocol set operation to set the status column
to `active'.  If the agent has sufficient information to
make the conceptual row available for use by the managed
device, the management protocol set operation succeeds (a
`noError' response is returned).  Otherwise, the management
protocol set operation fails with an error of
`inconsistentValue'.


                         NOTE WELL

     A conceptual row having a status column with value
     `notInService' or `notReady' is unavailable to the
     managed device.  As such, it is possible for the
     managed device to create its own instances during the
     time between the management protocol set operation
     which sets the status column to `createAndWait' and the
     management protocol set operation which sets the status
     column to `active'.  In this case, when the management
     protocol set operation is issued to set the status
     column to `active', the values held in the agent
     supersede those used by the managed device.

If the management station is prevented from setting the
status column to `active' (e.g., due to management station
or network failure) the conceptual row will be left in the
`notInService' or `notReady' state, consuming resources
```

```
            indefinitely.  The agent must detect conceptual rows that
            have been in either state for an abnormally long period of
            time and remove them.  It is the responsibility of the
            DESCRIPTION clause of the status column to indicate what an
            abnormally long period of time would be.  This period of
            time should be long enough to allow for human response time
            (including `think time') between the creation of the
            conceptual row and the setting of the status to `active'.
            In the absence of such information in the DESCRIPTION
            clause, it is suggested that this period be approximately 5
            minutes in length.  This removal action applies not only to
            newly-created rows, but also to previously active rows which
            are set to, and left in, the notInService state for a
            prolonged period exceeding that which is considered normal
            for such a conceptual row.


                             Conceptual Row Suspension

            When a conceptual row is `active', the management station
            may issue a management protocol set operation which sets the
            instance of the status column to `notInService'.  If the
            agent is unwilling to do so, the set operation fails with an
            error of `wrongValue'.  Otherwise, the conceptual row is
            taken out of service, and a `noError' response is returned.
            It is the responsibility of the DESCRIPTION clause of the
            status column to indicate under what circumstances the
            status column should be taken out of service (e.g., in order
            for the value of some other column of the same conceptual
            row to be modified).


                              Conceptual Row Deletion

            For deletion of conceptual rows, a management protocol set
            operation is issued which sets the instance of the status
            column to `destroy'.  This request may be made regardless of
            the current value of the status column (e.g., it is possible
            to delete conceptual rows which are either `notReady',
            `notInService' or `active'.)  If the operation succeeds,
            then all instances associated with the conceptual row are
            immediately removed."


    SYNTAX       INTEGER {
                     -- the following two values are states:
                     -- these values may be read or written
                     active(1),
                     notInService(2),

                     -- the following value is a state:
                     -- this value may be read, but not written
                     notReady(3),

                     -- the following three values are
                     -- actions: these values may be written,
                     --   but are never read
                     createAndGo(4),
                     createAndWait(5),
                     destroy(6)
                 }
```

2 Reference Manual

A multilingual Simple Network Management Protocol application featuring an Extensible Agent, simple manager, a MIB compiler and facilities for implementing SNMP MIBs etc.

snmp

Erlang module

The module `snmp` contains interface functions to the SNMP toolkit.

Common Data Types

The following data-types are used in the functions below:

- `datetime() = {date(), time()}`

  See *calendar* for more info.

Exports

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

A simple interactive configuration tool. Simple configuration files can be generated, but more complex configurations still have to be edited manually.

The tool is a textual based tool that asks some questions and generates `sys.config` and `*.conf` files.

Note that if the application shall support version 3, then the crypto app must be started before running this function (password generation).

Note also that some of the configuration files for the agent and manager share the same names. This means that they have to be stored in *different* directories!

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

`start(Type) -> ok | {error, Reason}`

Types:

Type = start_type()

Starts the SNMP application.

See *application* for more info.

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

`start_agent(Type) -> ok | {error, Reason}`

Types:

Type = start_type()

The SNMP application consists of several entities, of which the agent is one. This function starts the agent entity of the application.

Note that the only way to actually start the agent in this way is to add the agent related config after starting the application (e.g it cannot be part of the normal application config; sys.config). This is done by calling: `application:set_env(snmp, agent, Conf)`.

The default value for `Type` is `normal`.

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

`start_manager(Type) -> ok | {error, Reason}`

Types:

Type = start_type()

The SNMP application consists of several entities, of which the manager is one. This function starts the manager entity of the application.

Note that the only way to actually start the manager in this way is to add the manager related config after starting the application (e.g it cannot be part of the normal application config; sys.config). This is done by calling: `application:set_env(snmp, manager, Conf)`.

The default value for `Type` is `normal`.

`date_and_time() -> DateAndTime`

Types:

DateAndTime = [int()]

Returns current date and time as the data type DateAndTime, as specified in RFC1903. This is an OCTET STRING.

`date_and_time_to_universal_time_dst(DateAndTime) -> [utc()]`

Types:

DateAndTime = [int()]

utc() = {{Y,Mo,D},{H,M,S}}

Converts a DateAndTime list to a list of possible universal time(s). The universal time value on the same format as defined in calendar(3).

`date_and_time_to_string(DateAndTime) -> string()`

`date_and_time_to_string(DateAndTime, Validate) -> string()`

Types:

DateAndTime = [int()]

Validate = fun(Kind, Data) -> boolean()

Converts a DateAndTime list to a printable string, according to the DISPLAY-HINT definition in RFC2579.

The validation fun, `Validate`, allows for a more "flexible" validation of the `DateAndTime` argument. Whenever the data is found to not follow RFC2579, the fun is called to allow a more "lax" validation. See the *validate_date_and_time/2* function for more info on the `Validate` fun.

`date_and_time_to_string2(DateAndTime) -> string()`

Types:

DateAndTime = [int()]

Converts a DateAndTime list to a printable string, according to the DISPLAY-HINT definition in RFC2579, with the extension that it also allows the values "hours from UTC" = 14 together with "minutes from UTC" = 0.

`local_time_to_date_and_time_dst(Local) -> [DateAndTime]`

Types:

Local = {{Y,Mo,D},{H,M,S}}

DateAndTime = [int()]

Converts a local time value to a list of possible DateAndTime list(s). The local time value on the same format as defined in calendar(3).

universal_time_to_date_and_time(UTC) -> DateAndTime

Types:

UTC = {{Y,Mo,D},{H,M,S}}

DateAndTime = [int()]

Converts a universal time value to a DateAndTime list. The universal time value on the same format as defined in calendar(3).

validate_date_and_time(DateAndTime) -> bool()

validate_date_and_time(DateAndTime, Validate) -> bool()

Types:

DateAndTime = term()

Validate = fun(Kind, Data) -> boolean()

Checks if `DateAndTime` is a correct DateAndTime value, as specified in RFC2579. This function can be used in instrumentation functions to validate a DateAndTime value.

The validation fun, `Validate`, allows for a more "flexible" validation of the `DateAndTime` argument. Whenever the data is found to not follow RFC2579, the fun is called to allow a more "lax" validation. The input to the validation fun looks like this:

```
        Kind              Data
        --------------    ----------------------
        year              {Year1, Year2}
        month             Month
        day               Day
        hour              Hour
        minute            Minute
        seconds           Seconds
        deci_seconds      DeciSeconds
        diff              [Sign, Hour, Minute]
        valid_date        {Year, Month, Day}
```

passwd2localized_key(Alg, Passwd, EngineID) -> Key

Types:

Alg = algorithm()

algorithm() = md5 | sha

Passwd = string()

EngineID = string()

Key = list()

Generates a key that can be used as an authentication or privacy key using MD5 och SHA. The key is localized for EngineID.

`octet_string_to_bits(S) -> Val`

Types:

Val = bits()

Utility function for converting a value of type `OCTET-STRING` to `BITS`.

`bits_to_octet_string(B) -> Val`

Types:

Val = octet_string()

Utility function for converting a value of type `BITS` to `OCTET-STRING`.

`read_mib(FileName) -> {ok, mib()} | {error, Reason}`

Types:

FileName = string()

mib() = #mib{}

Reason = term()

Read a compiled mib.

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile, Start) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile, Start, Stop) -> ok | {error, Reason}`

Types:

LogDir = string()

Mibs = [MibName]

OutFile = string()

MibName = string()

LogName = string()

LogFile = string()

Start = Stop = null | datetime() | {local_time,datetime()} | {universal_time,datetime()}

Reason = term()

Converts an Audit Trail Log to a readable text file, where each item has a trailing TAB character, and any TAB character in the body of an item has been replaced by ESC TAB.

The function can be used on a running system, or by copying the entire log directory and calling this function. SNMP must be running in order to provide MIB information.

`LogDir` is the name of the directory where the audit trail log is stored. `Mibs` is a list of Mibs to be used. The function uses the information in the Mibs to convert for example object identifiers to their symbolic name. `OutFile` is the name of the generated text-file. `LogName` is the name of the log, `LogFile` is the name of the log file. `Start` is the start (first) date and time from which log events will be converted and `Stop` is the stop (last) date and time to which log events will be converted.

The format of an audit trail log text item is as follows:

```
Tag Addr - Community [TimeStamp] Vsn
```

```
PDU
```

where `Tag` is `request`, `response`, `report`, `trap` or `inform`; Addr is `IP:Port` (or comma space separated list of such); `Community` is the community parameter (SNMP version v1 and v2), or `SecLevel:"AuthEngineID":"UserName"` (SNMP v3); `TimeStamp` is a date and time stamp, and `Vsn` is the SNMP version. `PDU` is a textual version of the protocol data unit. There is a new line between `Vsn` and `PDU`.

change_log_size(LogName, NewSize) -> ok | {error, Reason}

Types:

LogName = string()

NewSize = {MaxBytes, MaxFiles}

MaxBytes = integer()

MaxFiles = integer()

Reason = term()

Changes the log size of the Audit Trail Log. The application must be configured to use the audit trail log function. Please refer to disk_log(3) in Kernel Reference Manual for a description of how to change the log size.

The change is permanent, as long as the log is not deleted. That means, the log size is remembered across reboots.

print_version_info() -> void()

print_version_info(Prefix) -> void()

Types:

Prefix = string() | integer()

Utility function(s) to produce a formatted printout of the versions info generated by the `versions1` function

This is the same as doing, e.g.:

```
        {ok, V} = snmp:versions1(),
        snmp:print_versions(V).
```

versions1() -> {ok, Info} | {error, Reason}

versions2() -> {ok, Info} | {error, Reason}

Types:

Info = [info()]

info() = term()

Reason = term()

Utility functions used to retrieve some system and application info.

The difference between the two functions is in how they get the modules to check. `versions1` uses the app-file and `versions2` uses the function `application:get_key`.

`print_versions(VersionInfo) -> void()`

`print_versions(Prefix, VersionInfo) -> void()`

Types:

VersionInfo = [version_info()]

version_info() = term()

Prefix = string() | integer()

Utility function to produce a formatted printout of the versions info generated by the `versions1` and `versions2` functions

Example:

```
        {ok, V} = snmp:versions1(),
        snmp:print_versions(V).
```

`enable_trace() -> void()`

Starts a dbg tracer that prints trace events to stdout (using plain io:format after a minor formatting).

`disable_trace() -> void()`

Stop the tracer.

`set_trace(Targets) -> void()`

Types:

Targets = target() | targets()

target() = module()

module() = atom()

targets() = [target() | {target(), target_options()}]

target_options() = [target_option()]

target_option() = {return_trace, boolean()} | {scope, scope()}

scope() = all_functions | exported_functions | function_name() | {function_name(), function_arity()}

function_name() = atom()

function_arity() = integer() >= 0

This function is used to set up default trace on function(s) for the given module or modules. The scope of the trace will be all *exported* functions (both the call info and the return value). Timestamp info will also be included.

`reset_trace(Targets) -> void()`

Types:

Targets = module() | modules()

modules() = [module()]

module() = atom()

This function is used to reset (disable) trace for the given module(s).

`set_trace(Targets, Opts) -> void()`

Types:

Targets = target() | targets()

target() = module()

module() = atom()

targets() = [target() | {target(), target_options()}]

target_options() = [target_option()]

target_option() = {return_trace, boolean()} | {scope, scope()}

scope() = all_functions | exported_functions | function_name() | {function_name(), function_arity()}

function_name() = atom()

function_arity() = integer() >= 0

Opts = disable | trace_options()

trace_options() = [trace_option()]

trace_option() = {timestamp, boolean()} | target_option()

This function is used to set up trace on function(s) for the given module or modules.

The example below sets up trace on the exported functions (default) of module `snmp_generic` and all functions of module `snmp_generic_mnesia`. With return values (which is default) and timestamps in both cases (which is also default):

```
snmp:enable_trace(),
snmp:set_trace([snmp_generic,
                        {snmp_generic_mnesia, [{scope, all_functions}]}]),
.
.
.
       snmp:set_trace(snmp_generic, disable),
.
.
.
snmp:disable_trace(),
```

See Also

calendar(3)

snmp

Application

This chapter describes the `snmp` application in OTP. The SNMP application provides the following services:

- a multilingual extensible SNMP agent
- a SNMP manager
- a MIB compiler

Configuration

The following configuration parameters are defined for the SNMP application. Refer to application(3) for more information about configuration parameters.

The snmp part of the config file specifying the configuration parameters is basically the following tuple:

```
{snmp, snmp_components_config()}
```

A minimal config file for starting a node with both a manager and an agent:

```
[{snmp,
  [{agent, [{db_dir, "/tmp/snmp/agent/db"},
            {config, [{dir, "/tmp/snmp/agent/conf"}]}]},
   {manager, [{config, [{dir, "/tmp/snmp/manager/conf"},
                        {db_dir, "/tmp/snmp/manager/db"}]}]}]}
  ]
 }
].
```

Each snmp component has its own set of configuration parameters, even though some of the types are common to both components.

```
snmp_components_config() -> [snmp_component_config()]
snmp_component_config() -> {agent, agent_options()} | {manager, manager_options()}
agent_options() = [agent_option()]
agent_option() = {restart_type,     restart_type()}     |
                 {agent_type,       agent_type()}       |
                 {agent_verbosity,  verbosity()}        |
                 {discovery,        agent_discovery()}  |
                 {versions,         versions()}         |
                 {priority,         priority()}         |
                 {multi_threaded,   multi_threaded()}   |
                 {db_dir,           db_dir()}           |
                 {db_init_error,    db_init_error()}    |
                 {local_db,         local_db()}         |
                 {net_if,           agent_net_if()}     |
                 {mibs,             mibs()}             |
                 {mib_storage,      mib_storage()}      |
                 {mib_server,       mib_server()}       |
                 {audit_trail_log,  audit_trail_log()}  |
                 {error_report_mod, error_report_mod()} |
                 {note_store,       note_store()}        |
                 {symbolic_store,   symbolic_store()}    |
                 {target_cache,     target_cache()}      |
                 {config,           agent_config()}
manager_options() = [manager_option()]
manager_option() = {restart_type,             restart_type()}     |
                   {net_if,                   manager_net_if()}   |
                   {server,                   server()}           |
                   {note_store,               note_store()}       |
                   {config,                   manager_config()}   |
                   {inform_request_behaviour, manager_irb()}      |
                   {mibs,                     manager_mibs()}     |
                   {priority,                 priority()}         |
                   {audit_trail_log,          audit_trail_log()}  |
                   {versions,                 versions()}         |
                   {def_user_mod,             def_user_module()   |
                   {def_user_data,            def_user_data()}
```

Agent specific config options and types:

`agent_type() = master | sub <optional>`

If `master`, one master agent is started. Otherwise, no agents are started.

Default is `master`.

`agent_discovery() = [agent_discovery_opt()] <optional>`

`agent_discovery_opt() = {terminating, agent_terminating_discovery_opts()} | {originating, agent_originating_discovery_opts()}`

The `terminating` options effects discovery initiated by a manager.

The `originating` options effects discovery initiated by this agent.

For defaults see the options in `agent_discovery_opt()`.

`agent_terminating_discovery_opts() = [agent_terminating_discovery_opt()] <optional>`

`agent_terminating_discovery_opt() = {enable, boolean()} | {stage2, discovery | plain} | {trigger_username, string()}`

These are options effecting discovery `terminating` in this agent (i.e. initiated by a manager).

The default values for the `terminating` discovery options are:

- enable: `true`
- stage2: `discovery`
- trigger_username: `""`

`agent_originating_discovery_opts() = [agent_originating_discovery_opt()] <optional>`

`agent_originating_discovery_opt() = {enable, boolean()}`

These are options effecting discovery `originating` in this agent.

The default values for the `originating` discovery options are:

- enable: `true`

`multi_threaded() = bool() <optional>`

If `true`, the agent is multi-threaded, with one thread for each get request.

Default is `false`.

`db_dir() = string() <mandatory>`

Defines where the SNMP agent internal db files are stored.

`local_db() = [local_db_opt()] <optional>`

`local_db_opt() = {repair, agent_repair()} | {auto_save, agent_auto_save()} | {verbosity, verbosity()}`

Defines options specific for the SNMP agent local database.

For defaults see the options in `local_db_opt()`.

`agent_repair() = false | true | force <optional>`

When starting snmpa_local_db it always tries to open an existing database. If `false`, and some errors occur, a new database is created instead. If `true`, an existing file will be repaired. If `force`, the table will be repaired even if it was properly closed.

Default is `true`.

`agent_auto_save() = integer() | infinity <optional>`

The auto save interval. The table is flushed to disk whenever not accessed for this amount of time.

Default is `5000`.

`agent_net_if() = [agent_net_if_opt()] <optional>`

`agent_net_if_opt() = {module, agent_net_if_module()} | {verbosity, verbosity()} | {options, agent_net_if_options()}`

Defines options specific for the SNMP agent network interface entity.

For defaults see the options in `agent_net_if_opt()`.

`agent_net_if_module() = atom() <optional>`

Module which handles the network interface part for the SNMP agent. Must implement the *snmpa_network_interface* behaviour.

Default is `snmpa_net_if`.

`agent_net_if_options() = [agent_net_if_option()] <optional>`

`agent_net_if_option() = {bind_to, bind_to()} | {sndbuf, sndbuf()} | {recbuf, recbuf()} | {no_reuse, no_reuse()} | {req_limit, req_limit()} | {filter, agent_net_if_filter_options()}`

These options are actually specific to the used module. The ones shown here are applicable to the default `agent_net_if_module()`.

For defaults see the options in `agent_net_if_option()`.

`req_limit() = integer() | infinity <optional>`

Max number of simultaneous requests handled by the agent.

Default is `infinity`.

`agent_net_if_filter_options() = [agent_net_if_filter_option()] <optional>`

`agent_net_if_filter_option() = {module, agent_net_if_filter_module()}`

These options are actually specific to the used module. The ones shown here are applicable to the default `agent_net_if_filter_module()`.

For defaults see the options in `agent_net_if_filter_option()`.

`agent_net_if_filter_module() = atom() <optional>`

Module which handles the network interface filter part for the SNMP agent. Must implement the *snmpa_network_interface_filter* behaviour.

Default is `snmpa_net_if_filter`.

`agent_mibs() = [string()] <optional>`

Specifies a list of MIBs (including path) that defines which MIBs are initially loaded into the SNMP master agent.

Note that the following mibs will always be loaded:

- version v1: `STANDARD-MIB`
- version v2: `SNMPv2`
- version v3: `SNMPv2`, `SNMP-FRAMEWORK-MIB` and `SNMP-MPD-MIB`

Default is `[]`.

`mib_storage() = ets | {ets, Dir} | {ets, Dir, Action} | dets | {dets, Dir} | {dets, Dir, Action} | mnesia | {mnesia, Nodes} | {mnesia, Nodes, Action} <optional>`

Specifies how info retrieved from the mibs will be stored.

If `mib_storage` is `{ets, Dir}`, the table will also be stored on file. If `Dir` is `default`, then `db_dir` will be used.

If `mib_storage` is `dets` or if `Dir` is `default`, then `db_dir` will be used for `Dir`.

If `mib_storage` is `mnesia` then `erlang:nodes()` will be used for `Nodes`.

Default is `ets`.

`Dir = default | string()`. Dir is the directory where the files will be stored. If `default`, then `db_dir` will be used.

`Nodes = visible | connected | [node()]`. `Nodes = visible` is translated to `erlang:nodes(visible)`. `Nodes = connected` is translated to `erlang:nodes(connected)`. If `Nodes = []` then the own node is assumed.

`Action = clear | keep`. Default is `keep`. `Action` is used to specify what shall be done if the mnesia/dets table already exist.

`mib_server() = [mib_server_opt()] <optional>`

`mib_server_opt() = {mibentry_override, mibentry_override()} | {trapentry_override, trapentry_override()} | {verbosity, verbosity()} | {cache, mibs_cache()}`

Defines options specific for the SNMP agent mib server.

For defaults see the options in `mib_server_opt()`.

`mibentry_override() = bool() <optional>`

If this value is false, then when loading a mib each mib- entry is checked prior to installation of the mib. The purpose of the check is to prevent that the same symbolic mibentry name is used for different oid's.

Default is `false`.

`trapentry_override() = bool() <optional>`

If this value is false, then when loading a mib each trap is checked prior to installation of the mib. The purpose of the check is to prevent that the same symbolic trap name is used for different trap's.

Default is `false`.

`mibs_cache() = bool() | mibs_cache_opts() <optional>`

Shall the agent utilize the mib server lookup cache or not.

Default is `true` (in which case the `mibs_cache_opts()` default values apply).

`mibs_cache_opts() = [mibs_cache_opt()] <optional>`

`mibs_cache_opt() = {autogc, mibs_cache_autogc()} | {gclimit, mibs_cache_gclimit()} | {age, mibs_cache_age()}`

Defines options specific for the SNMP agent mib server cache.

For defaults see the options in `mibs_cache_opt()`.

`mibs_cache_autogc() = bool() <optional>`

Defines if the mib server shall perform cache gc automatically or leave it to the user (see *gc_mibs_cache/0,1,2,3*).

Default is `true`.

`mibs_cache_age() = integer() > 0 <optional>`

Defines how old the entries in the cache will be allowed before they are GC'ed (assuming GC is performed). Each entry in the cache is "touched" whenever it is accessed.

The age is defined in milliseconds.

Default is `10 timutes`.

`mibs_cache_gclimit() = integer() > 0 | infinity <optional>`

When performing a GC, this is the max number of cache entries that will be deleted from the cache.

The reason for having this limit is that if the cache is large, the GC can potentially take a long time, during which the agent is locked.

Default is `100`.

`error_report_mod() = atom() <optional>`

Defines an error report module, implementing the *snmpa_error_report* behaviour. Two modules are provided with the toolkit: `snmpa_error_logger` and `snmpa_error_io`.

Default is `snmpa_error_logger`.

`symbolic_store() = [symbolic_store_opt()]`

`symbolic_store_opt() = {verbosity, verbosity()}`

Defines options specific for the SNMP agent symbolic store.

For defaults see the options in `symbolic_store_opt()`.

`target_cache() = [target_cache_opt()]`

`target_cache_opt() = {verbosity, verbosity()}`

Defines options specific for the SNMP agent target cache.

For defaults see the options in `target_cache_opt()`.

`agent_config() = [agent_config_opt()] <mandatory>`

`agent_config_opt() = {dir, agent_config_dir()} | {force_load, force_load()} | {verbosity, verbosity()}`

Defines specific config related options for the SNMP agent.

For defaults see the options in `agent_config_opt()`.

`agent_config_dir = dir() <mandatory>`

Defines where the SNMP agent configuration files are stored.

`force_load() = bool() <optional>`

If `true` the configuration files are re-read during start-up, and the contents of the configuration database ignored. Thus, if `true`, changes to the configuration database are lost upon reboot of the agent.

Default is `false`.

Manager specific config options and types:

`server() = [server_opt()] <optional>`

`server_opt() = {timeout, server_timeout()} | {verbosity, verbosity()}`

Specifies the options for the manager server process.

Default is `silence`.

`server_timeout() = integer() <optional>`

Asynchroneous request cleanup time. For every requests, some info is stored internally, in order to be able to deliver the reply (when it arrives) to the proper destination. If the reply arrives, this info will be deleted. But if there is no reply (in time), the info has to be deleted after the *best before* time has been passed. This cleanup will be performed at regular intervals, defined by the `server_timeout()` time. The information will have an *best before* time, defined by the `Expire` time given when calling the request function (see *async_get*, *async_get_next* and *async_set*).

Time in milli-seconds.

Default is `30000`.

`manager_config() = [manager_config_opt()] <mandatory>`

`manager_config_opt() = {dir, manager_config_dir()} | {db_dir, manager_db_dir()} | {db_init_error, db_init_error()} | {repair, manager_repair()} | {auto_save, manager_auto_save()} | {verbosity, verbosity()}`

Defines specific config related options for the SNMP manager.

For defaults see the options in `manager_config_opt()`.

`manager_config_dir = dir() <mandatory>`

Defines where the SNMP manager configuration files are stored.

`manager_db_dir = dir() <mandatory>`

Defines where the SNMP manager store persistent data.

`manager_repair() = false | true | force <optional>`

Defines the repair option for the persistent database (if and how the table is repaired when opened).

Default is `true`.

`manager_auto_save() = integer() | infinity <optional>`

The auto save interval. The table is flushed to disk whenever not accessed for this amount of time.

Default is `5000`.

`manager_irb() = auto | user | {user, integer()} <optional>`

This option defines how the manager will handle the sending of response (acknowledgment) to received inform-requests.

- `auto` - The manager will autonomously send response (acknowledgment> to inform-request messages.
- `{user, integer()}` - The manager will send response (acknowledgment) to inform-request messages when the *handle_inform* function completes. The integer is the time, in milli-seconds, that the manager will consider the stored inform-request info valid.
- `user` - Same as `{user, integer()}`, except that the default time, 15 seconds (15000), is used.

See *snmpm_network_interface*, *handle_inform* and *definition of the manager net if* for more info.

Default is `auto`.

`manager_mibs() = [string()] <optional>`

Specifies a list of MIBs (including path) and defines which MIBs are initially loaded into the SNMP manager.

Default is `[]`.

`manager_net_if() = [manager_net_if_opt()] <optional>`

`manager_net_if_opt() = {module, manager_net_if_module()} | {verbosity, verbosity()} | {options, manager_net_if_options()}`

Defines options specific for the SNMP manager network interface entity.

For defaults see the options in `manager_net_if_opt()`.

`manager_net_if_options() = [manager_net_if_option()] <optional>`

`manager_net_if_option() = {bind_to, bind_to()} | {sndbuf, sndbuf()} | {recbuf, recbuf()} | {no_reuse, no_reuse()} | {filter, manager_net_if_filter_options()}`

These options are actually specific to the used module. The ones shown here are applicable to the default `manager_net_if_module()`.

For defaults see the options in `manager_net_if_option()`.

`manager_net_if_module() = atom() <optional>`

Module which handles the network interface part for the SNMP manager. Must implement the *snmpm_network_interface* behaviour.

Default is `snmpm_net_if`.

`manager_net_if_filter_options() = [manager_net_if_filter_option()] <optional>`

`manager_net_if_filter_option() = {module, manager_net_if_filter_module()}`

These options are actually specific to the used module. The ones shown here are applicable to the default `manager_net_if_filter_module()`.

For defaults see the options in `manager_net_if_filter_option()`.

`manager_net_if_filter_module() = atom() <optional>`

Module which handles the network interface filter part for the SNMP manager. Must implement the *snmpm_network_interface_filter* behaviour.

Default is `snmpm_net_if_filter`.

`def_user_module() = atom() <optional>`

The module implementing the default user. See the *snmpm_user* behaviour.

Default is `snmpm_user_default`.

`def_user_data() = term() <optional>`

Data for the default user. Passed to the user module when calling the callback functions.

Default is `undefined`.

Common config types:

`restart_type() = permanent | transient | temporary`

See *supervisor* documentation for more info.

Default is `permanent` for the agent and `transient` for the manager.

`db_init_error() = terminate | create`

Defines what to do if the agent or manager is unable to open an existing database file. `terminate` means that the agent/manager will terminate and `create` means that the agent/manager will remove the faulty file(s) and create new ones.

Default is `terminate`.

`priority() = atom() <optional>`

Defines the Erlang priority for all SNMP processes.

Default is `normal`.

`versions() = [version()] <optional>`

`version() = v1 | v2 | v3`

Which SNMP versions shall be accepted/used.

Default is `[v1,v2,v3]`.

`verbosity() = silence | info | log | debug | trace <optional>`

Verbosity for a SNMP process. This specifies now much debug info is printed.

Default is `silence`.

`bind_to() = bool() <optional>`

If `true`, net_if binds to the IP address. If `false`, net_if listens on any IP address on the host where it is running.

Default is `false`.

`no_reuse() = bool() <optional>`

If `true`, net_if does not specify that the IP and port address should be reusable. If `false`, the address is set to reusable.

Default is `false`.

`recbuf() = integer() <optional>`

Receive buffer size.

Default value is defined by `gen_udp`.

`sndbuf() = integer() <optional>`

Send buffer size.

Default value is defined by `gen_udp`.

note_store() = [note_store_opt()] <optional>

note_store_opt() = {timeout, note_store_timeout()} | {verbosity, verbosity()}

Specifies the start-up verbosity for the SNMP note store.

For defaults see the options in note_store_opt().

note_store_timeout() = integer() <optional>

Note cleanup time. When storing a note in the note store, each note is given lifetime. Every timeout the note_store process performs a GC to remove the expired note's. Time in milli-seconds.

Default is 30000.

audit_trail_log() = [audit_trail_log_opt()] <optional>

audit_trail_log_opt() = {type, atl_type()} | {dir, atl_dir()} | {size, atl_size()} | {repair, atl_repair()} | {seqno, atl_seqno()}

If present, this option specifies the options for the audit trail logging. The disk_log module is used to maintain a wrap log. If present, the dir and size options are mandatory.

If not present, audit trail logging is not used.

atl_type() = read | write | read_write <optional>

Specifies what type of an audit trail log should be used. The effect of the type is actually different for the the agent and the manager.

For the agent:

- If write is specified, only set requests are logged.
- If read is specified, only get requests are logged.
- If read_write, all requests are logged.

For the manager:

- If write is specified, only sent messages are logged.
- If read is specified, only received messages are logged.
- If read_write, both outgoing and incoming messages are logged.

Default is read_write.

atl_dir = dir() <mandatory>

Specifies where the audit trail log should be stored.

If audit_trail_log specifies that logging should take place, this parameter *must* be defined.

atl_size() = {integer(), integer()} <mandatory>

Specifies the size of the audit trail log. This parameter is sent to disk_log.

If audit_trail_log specifies that logging should take place, this parameter *must* be defined.

atl_repair() = true | false | truncate | snmp_repair <optional>

Specifies if and how the audit trail log shall be repaired when opened. Unless this parameter has the value snmp_repair it is sent to disk_log. If, on the other hand, the value is snmp_repair, snmp attempts to handle certain faults on its own. And even if it cannot repair the file, it does not truncate it directly, but instead *moves it aside* for later off-line analysis.

Default is true.

`atl_seqno() = true | false <optional>`

Specifies if the audit trail log entries will be (sequence) numbered or not. The range of the sequence numbers are according to RFC 5424, i.e. 1 through 2147483647.

Default is `false`.

See Also

application(3), disk_log(3)

snmpa

Erlang module

The module `snmpa` contains interface functions to the SNMP agent.

DATA TYPES

```
oid() = [byte()]
atl_type() = read | write | read_write
notification_delivery_info() = #snmpa_notification_delivery_info{}
```

The `oid()` type is used to represent an ASN.1 OBJECT IDENTIFIER.

The record `snmpa_notification_delivery_info` contains the following fields:

`tag = term()`

A user defined identity representing this notification send operation.

`mod = module()`

A module implementing the *snmpa_notification_delivery_info_receiver* behaviour. The info functions of this module will be called at various stages of delivery.

`extra = term()`

This is any extra info the user wants to have supplied when the functions in the callback module is called.

Exports

`add_agent_caps(SysORID, SysORDescr) -> SysORIndex`

Types:

SysORID = oid()

SysORDescr = string()

SysORIndex = integer()

This function can be used to add an AGENT-CAPABILITY statement to the sysORTable in the agent. The table is defined in the SNMPv2-MIB.

`del_agent_caps(SysORIndex) -> void()`

Types:

SysORIndex = integer()

This function can be used to delete an AGENT-CAPABILITY statement to the sysORTable in the agent. This table is defined in the SNMPv2-MIB.

`get_agent_caps() -> [[SysORIndex, SysORID, SysORDescr, SysORUpTime]]`

Types:

SysORIndex = integer()

SysORId = oid()

SysORDescr = string()

SysORUpTime = integer()

Returns all AGENT-CAPABILITY statements in the sysORTable in the agent. This table is defined in the SNMPv2-MIB.

`get(Agent, Vars) -> Values | {error, Reason}`

`get(Agent, Vars, Context) -> Values | {error, Reason}`

Types:

Agent = pid() | atom()

Vars = [oid()]

Context = string()

Values = [term()]

Reason = {atom(), oid()}

Performs a GET operation on the agent. All loaded MIB objects are visible in this operation. The agent calls the corresponding instrumentation functions just as if it was a GET request coming from a manager.

Note that the request specific parameters (such as *current_request_id*) are not accessible for the instrumentation functions if this function is used.

`get_next(Agent, Vars) -> Values | {error, Reason}`

`get_next(Agent, Vars, Context) -> Values | {error, Reason}`

Types:

Agent = pid() | atom()

Vars = [oid()]

Context = string()

Values = [{oid(), term()}]

Reason = {atom(), oid()}

Performs a GET-NEXT operation on the agent. All loaded MIB objects are visible in this operation. The agent calls the corresponding instrumentation functions just as if it was a GET request coming from a manager.

Note that the request specific parameters (such as `snmpa:current_request_id/0` are not accessible for the instrumentation functions if this function is used.

`backup(BackupDir) -> ok | {error, Reason}`

`backup(Agent, BackupDir) -> ok | {error, Reason}`

Types:

BackupDir = string()

Agent = pid() | atom()

Backup persistent/permanent data handled by the agent (such as local-db, mib-data and vacm).

Data stored by mnesia is not handled.

BackupDir cannot be identical to DbDir.

info() -> [{Key, Value}]

info(Agent) -> [{Key, Value}]

Types:

Agent = pid() | atom()

Returns a list (a dictionary) containing information about the agent. Information includes loaded MIBs, registered sub-agents, some information about the memory allocation.

As of version 4.4 the format of the info has been changed. To convert the info to the old format, call the *old_info_format* function.

old_info_format(NewInfo) -> OldInfo

Types:

OldInfo = NewInfo = [{Key, Value}]

As of version 4.4 the format of the info has been changed. This function is used to convert to the old (pre-4.4) info format.

load_mibs(Mibs) -> ok | {error, Reason}

load_mibs(Agent,Mibs) -> ok | {error, Reason}

Types:

Agent = pid() | atom()

Mibs = [MibName]

MibName = string()

Reason = term()

Loads `Mibs` into an agent. If the agent cannot load all MIBs, it will indicate where loading was aborted. The `MibName` is the name of the Mib, including the path to where the compiled mib is found. For example,

```
        Dir = code:priv_dir(my_app) ++ "/mibs/",
        snmpa:load_mibs(snmp_master_agent, [Dir ++ "MY-MIB"]).
```

unload_mibs(Mibs) -> ok | {error, Reason}

unload_mibs(Agent,Mibs) -> ok | {error, Reason}

Types:

Agent = pid() | atom()

Mibs = [MibName]

MibName = string()

Unloads MIBs into an agent. If it cannot unload all MIBs, it will indicate where unloading was aborted.

which_mibs() -> Mibs

which_mibs(Agent) -> Mibs

Types:

Agent = pid() | atom()

Mibs = [{MibName, MibFile}]

MibName = atom()

MibFile = string()

Retrieve the list of all the mibs loaded into this agent. Default is the master agent.

`whereis_mib(MibName) -> {ok, MibFile} | {error, Reason}`

`whereis_mib(Agent, MibName) -> {ok, MibFile} | {error, Reason}`

Types:

Agent = pid() | atom()

MibName = atom()

MibFile = string()

Reason = term()

Get the full path to the (compiled) mib-file.

`current_request_id() -> {value, RequestId} | false`

`current_context() -> {value, Context} | false`

`current_community() -> {value, Community} | false`

`current_address() -> {value, Address} | false`

Types:

RequestId = integer()

Context = string()

Community = string()

Address = term()

Get the request-id, context, community and address of the request currently being processed by the agent.

Note that these functions is intended to be called by the instrumentation functions and *only* if they are executed in the context of the agent process (e.g. it does not work if called from a spawned process).

`enum_to_int(Name, Enum) -> {value, Int} | false`

`enum_to_int(Db, Name, Enum) -> {value, Int} | false`

Types:

Db = term()

Name = atom()

Enum = atom()

Int = int()

Converts the symbolic value `Enum` to the corresponding integer of the enumerated object or type `Name` in a MIB. The MIB must be loaded.

`false` is returned if the object or type is not defined in any loaded MIB, or if it does not define the symbolic value as enumerated.

`Db` is a reference to the symbolic store database (retrieved by a call to `get_symbolic_store_db/0`).

`int_to_enum(Name, Int) -> {value, Enum} | false`

`int_to_enum(Db, Name, Int) -> {value, Enum} | false`

Types:

Db = term()

Name = atom()

Int = int()

Enum = atom()

Converts the integer `Int` to the corresponding symbolic value of the enumerated object or type `Name` in a MIB. The MIB must be loaded.

`false` is returned if the object or type is not defined in any loaded MIB, or if it does not define the symbolic value as enumerated.

`Db` is a reference to the symbolic store database (retrieved by a call to `get_symbolic_store_db/0`).

`name_to_oid(Name) -> {value, oid()} | false`

`name_to_oid(Db, Name) -> {value, oid()} | false`

Types:

Db = term()

Name = atom()

Looks up the OBJECT IDENTIFIER of a MIB object, given the symbolic name. Note, the OBJECT IDENTIFIER is given for the object, not for an instance.

`false` is returned if the object is not defined in any loaded MIB.

`Db` is a reference to the symbolic store database (retrieved by a call to `get_symbolic_store_db/0`).

`oid_to_name(OID) -> {value, Name} | false`

`oid_to_name(Db, OID) -> {value, Name} | false`

Types:

Db = term()

OID = oid()

Name = atom()

Looks up the symbolic name of a MIB object, given OBJECT IDENTIFIER.

`false` is returned if the object is not defined in any loaded MIB.

`Db` is a reference to the symbolic store database (retrieved by a call to `get_symbolic_store_db/0`).

`which_aliasnames() -> Result`

Types:

Result = [atom()]

Retrieve all alias-names known to the agent.

`which_tables() -> Result`

Types:

Result = [atom()]

Retrieve all tables known to the agent.

`which_variables() -> Result`

Types:

Result = [atom()]

Retrieve all variables known to the agent.

`which_notifications() -> Result`

Types:

Result = [{Name, MibName, Info}]

Name = atom()

MibName = atom()

Info = term()

Retrieve all notifications (and traps) known to the agent.

`log_to_txt(LogDir, Mibs)`

`log_to_txt(LogDir, Mibs, OutFile) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile, Start) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile, Start, Stop) -> ok | {error, Reason}`

Types:

LogDir = string()

Mibs = [MibName]

MibName = string()

OutFile = string()

LogName = string()

LogFile = string()

Start = Stop = null | datetime() | {local_time,datetime()} | {universal_time,datetime()}

Reason = disk_log_open_error() | file_open_error() | term()

disk_log_open_error() = {LogName, term()}

file_open_error() = {OutFile, term()}

Converts an Audit Trail Log to a readable text file. `OutFile` defaults to "./snmpa_log.txt". `LogName` defaults to "snmpa_log". `LogFile` defaults to "snmpa.log". See *snmp:log_to_txt* for more info.

`change_log_size(NewSize) -> ok | {error, Reason}`

Types:

NewSize = {MaxBytes, MaxFiles}

MaxBytes = integer()

MaxFiles = integer()

Reason = term()

Changes the log size of the Audit Trail Log. The application must be configured to use the audit trail log function. Please refer to disk_log(3) in Kernel Reference Manual for a description of how to change the log size.

The change is permanent, as long as the log is not deleted. That means, the log size is remembered across reboots.

`set_log_type(NewType) -> {ok, OldType} | {error, Reason}`

`set_log_type(Agent, NewType) -> {ok, OldType} | {error, Reason}`

Types:

NewType = OldType = atl_type()

Agent = pid() | atom()

Reason = term()

Changes the run-time Audit Trail log type.

Note that this has no effect on the application configuration as defined by configuration files, so a node restart will revert the config to whatever is in those files.

This function is primarily useful in testing/debugging scenarios.

`mib_of(Oid) -> {ok, MibName} | {error, Reason}`

`mib_of(Agent, Oid) -> {ok, MibName} | {error, Reason}`

Types:

Agent = pid() | atom()

Oid = oid()

MibName = atom()

Reason = term()

Finds the mib corresponding to the `Oid`. If it is a variable, the Oid must be <Oid for var>.0 and if it is a table, Oid must be <table>.<entry>.<col>.<any>

`me_of(Oid) -> {ok, Me} | {error, Reason}`

`me_of(Agent, Oid) -> {ok, Me} | {error, Reason}`

Types:

Agent = pid() | atom()

Oid = oid()

Me = #me{}

Reason = term()

Finds the mib entry corresponding to the `Oid`. If it is a variable, the Oid must be <Oid for var>.0 and if it is a table, Oid must be <table>.<entry>.<col>.<any>

`invalidate_mibs_cache() -> void()`

`invalidate_mibs_cache(Agent) -> void()`

Types:

Agent = pid() | atom()

Invalidate the mib server cache.

The entire contents of the cache will be deleted.

`enable_mibs_cache() -> void()`

`enable_mibs_cache(Agent) -> void()`

Types:

Agent = pid() | atom()

Enable the mib server cache.

`disable_mibs_cache() -> void()`

`disable_mibs_cache(Agent) -> void()`

Types:

Agent = pid() | atom()

Disable the mib server cache.

`which_mibs_cache_size() -> void()`

`which_mibs_cache_size(Agent) -> void()`

Types:

Agent = pid() | atom()

Retreive the size of the mib server cache.

`gc_mibs_cache() -> {ok, NumElementsGCed} | {error, Reason}`

`gc_mibs_cache(Agent) -> {ok, NumElementsGCed} | {error, Reason}`

`gc_mibs_cache(Age) -> {ok, NumElementsGCed} | {error, Reason}`

`gc_mibs_cache(Agent, Age) -> {ok, NumElementsGCed} | {error, Reason}`

`gc_mibs_cache(Age, GcLimit) -> {ok, NumElementsGCed} | {error, Reason}`

`gc_mibs_cache(Agent, Age, GcLimit) -> {ok, NumElementsGCed} | {error, Reason}`

Types:

Agent = pid() | atom()

Age = integer() > 0

GcLimit = integer() > 0 | infinity

NumElementsGCed = integer() >= 0

Reason = term()

Perform mib server cache gc.

Manually performs a mib server cache gc. This can be done regardless of the value of the `autogc` option. The `NumElementsGCed` value indicates how many elements where actually removed from the cache.

`enable_mibs_cache_autogc() -> void()`

`enable_mibs_cache_autogc(Agent) -> void()`

Types:

Agent = pid() | atom()

Enable automatic gc of the mib server cache.

`disable_mibs_cache_autogc() -> void()`

`disable_mibs_cache_autogc(Agent) -> void()`

Types:

Agent = pid() | atom()

Disable automatic gc of the mib server cache.

`update_mibs_cache_age(NewAge) -> ok | {error, Reason}`

`update_mibs_cache_age(Agent, NewAge) -> ok | {error, Reason}`

Types:

Agent = pid() | atom()

NewAge = integer() > 0

Reason = term()

Change the mib server cache `age` property.

`update_mibs_cache_gclimit(NewGcLimit) -> ok | {error, Reason}`

`update_mibs_cache_gclimit(Agent, NewGCLimit) -> ok | {error, Reason}`

Types:

Agent = pid() | atom()

NewGcLimit = integer() > 0 | infinity

Reason = term()

Change the mib server cache `gclimit` property.

register_notification_filter(Id, Mod, Data) -> ok | {error, Reason}

register_notification_filter(Agent, Id, Mod, Data) -> ok | {error, Reason}

register_notification_filter(Id, Mod, Data, Where) -> ok | {error, Reason}

register_notification_filter(Agent, Id, Mod, Data, Where) -> ok | {error, Reason}

Types:

Agent = pid() | atom()

Id = filter_id()

filter_id() = term()

Mod = atom()

Data = term()

Where = filter_position()

Reason = term()

filter_position() = first | last | {insert_before, filter_id()} | {insert_after, filter_id()}

Registers a notification filter.

`Mod` is a module implementing the `snmpa_notification_filter` behaviour.

`Data` will be passed on to the filter when calling the functions of the behaviour.

unregister_notification_filter(Id) -> ok | {error, Reason}

unregister_notification_filter(Agent, Id) -> ok | {error, Reason}

Types:

Agent = pid() | atom()

Id = filter_id()

filter_id() = term()

Unregister a notification filter.

which_notification_filter() -> Filters

which_notification_filter(Agent) -> Filters

Types:

Agent = pid() | atom()

Filters = [filter_id()]

filter_id() = term()

List all notification filters in an agent.

set_request_limit(NewLimit) -> {ok, OldLimit} | {error, Reason}

set_request_limit(Agent, NewLimit) -> {ok, OldLimit} | {error, Reason}

Types:

NewLimit = OldLimit = infinity | integer() >= 0

Agent = pid() | atom()

Reason = term()

Changes the request limit.

Note that this has no effect on the application configuration as defined by configuration files, so a node restart will revert the config to whatever is in those files.

This function is primarily useful in load regulation scenarios.

register_subagent(Agent, SubTreeOid, Subagent) -> ok | {error, Reason}

Types:

Agent = pid() | atom()

SubTreeOid = oid()

SubAgent = pid()

Registers a sub-agent under a sub-tree of another agent.

It is easy to make mistakes when registering sub-agents and this activity should be done carefully. For example, a strange behaviour would result from the following configuration:

```
snmp_agent:register_subagent(MAPid,[1,2,3,4],SA1),
snmp_agent:register_subagent(SA1,[1,2,3], SA2).
```

`SA2` will not get requests starting with object identifier `[1,2,3]` since `SA1` does not.

unregister_subagent(Agent, SubagentOidOrPid) -> ok | {ok, SubAgentPid} | {error, Reason}

Types:

Agent = pid() | atom()

SubTreeOidorPid = oid() | pid()

Unregister a sub-agent. If the second argument is a pid, then that sub-agent will be unregistered from all trees in `Agent`.

send_notification(Agent, Notification, Receiver)

send_notification(Agent, Notification, Receiver, Varbinds)

send_notification(Agent, Notification, Receiver, NotifyName, Varbinds)

send_notification(Agent, Notification, Receiver, NotifyName, ContextName, Varbinds) -> void()

send_notification(Agent, Notification, Receiver, NotifyName, ContextName, Varbinds, LocalEngineID) -> void()

Types:

Agent = pid() | atom()

Notification = atom()

Receiver = no_receiver | {Tag, Recv} | notification_delivery_info()

Tag = term()

Recv = receiver()

receiver() = pid() | atom() | {Mod, Func, Args}

Mod = atom()

Func = atom()

Args = list()

NotifyName = string()

ContextName = string()

Varbinds = varbinds()

varbinds() = [varbind()]

varbind() = {Variable, Value} | {Column, RowIndex, Value} | {OID, Value}

Variable = atom()

Column = atom()

OID = oid()

Value = term()

RowIndex = [int()]

LocalEngineID = string()

Sends the notification `Notification` to the management targets defined for `NotifyName` in the `snmpNotifyTable` in SNMP-NOTIFICATION-MIB from the specified context. If no `NotifyName` is specified (or if it is `""`), the notification is sent to all management targets (`Addresses` below). If no `ContextName` is specified, the default `""` context is used.

The parameter `Receiver` specifies where information about delivery of Inform-Requests should be sent. The agent sends Inform-Requests and waits for acknowledgments from the managers. `Receiver` can have three values:

- `no_receiver` - No information is delivered.
- `{Tag, Recv}` - The information is delivered either via messages or via a function call according to the value of `Recv`.
- `notification_delivery_info()` - The information is delivered via a function call according to this data. See the *DATA TYPES* section above for details.

If `Receiver` has the value `{Tag, Recv}`, the delivery is done according to `Recv`:

- `pid() | atom()` - The info will be delivered in the following messages:
  - `{snmp_targets, Tag, Addresses}`

    This inform the user which target addresses the notification was sent to.
  - `{snmp_notification, Tag, {got_response, Address}}`

    This informs the user that this target address acknowledged the notification.
  - `{snmp_notification, Tag, {no_response, Address}}`

    This informs the user that this target address did not acknowledge notification.

  The notification is sent as an Inform-Request to each target address in `Addresses` and if there are no targets for which an Inform-Request is sent, `Addresses` is the empty list `[]`.

  The `receiver` will first be sent the `snmp_targets` message, and then for each address in `Addresses` list, one of the two `snmp_notification` messages.
- `{Mod, Func, Args}` - The info will be delivered via the function call:

`Mod:Func([Msg | Args])`

where `Msg` has the same content and purpose as the messages descrived above.

`Address` is a management target address and `Addresses` is a list of management target addresses. They are defined as followes:

```
Addresses  = [address()]
Address    = address()
address()  = v1_address() | v3_address()
v1_address() = {TDomain, TAddress}
v3_address() = {{TDomain, TAddress}, V3MsgData}
TDomain    = tdoamin()
TAddress   = taddress()
tdomain()  = The oid of snmpUDPDomain
             This is the only supported transport domain.
taddress() = [A1, A2, A3, A4, P1, P3]
             The 4 first bytes makes up the IP-address and the last 2,
             the UDP-port number.
V3MsgData  = v3_msg_data()
v3_msg_data() = term()
```

If `Receiver` is a `notification_delivery_info()` record, then the information about the notification delivery will be delivered to the `receiver` via the callback functions defined by the *snmpa_notification_delivery_info_receiver* behaviour according to the content of the `notification_delivery_info()` record.

The optional argument `Varbinds` defines values for the objects in the notification. If no value is given for an object, the `Agent` performs a get-operation to retrieve the value.

`Varbinds` is a list of `Varbind`, where each `Varbind` is one of:

- `{Variable, Value}`, where `Variable` is the symbolic name of a scalar variable referred to in the notification specification.
- `{Column, RowIndex, Value}`, where `Column` is the symbolic name of a column variable. `RowIndex` is a list of indices for the specified element. If this is the case, the OBJECT IDENTIFIER sent in the notification is the `RowIndex` appended to the OBJECT IDENTIFIER for the table column. This is the OBJECT IDENTIFIER which specifies the element.
- `{OID, Value}`, where `OID` is the OBJECT IDENTIFIER for an instance of an object, scalar variable, or column variable.

For example, to specify that `sysLocation` should have the value `"upstairs"` in the notification, we could use one of:

- `{sysLocation, "upstairs"}` or
- `{[1,3,6,1,2,1,1,6,0], "upstairs"}` or
- `{?sysLocation_instance, "upstairs"}` (provided that the generated `.hrl` file is included)

If a variable in the notification is a table element, the `RowIndex` for the element must be given in the `Varbinds` list. In this case, the OBJECT IDENTIFIER sent in the notification is the OBJECT IDENTIFIER that identifies this element. This OBJECT IDENTIFIER could be used in a get operation later.

This function is asynchronous, and does not return any information. If an error occurs, `user_err/2` of the error report module is called and the notification is discarded.

> Note:
>
> Note that the use of the LocalEngineID argument is only intended for special cases, if the agent is to "emulate" multiple EngineIDs! By default, the agent uses the value of `SnmpEngineID` (see SNMP-FRAMEWORK-MIB).

`discovery(TargetName, Notification) -> {ok, ManagerEngineID} | {error, Reason}`

`discovery(TargetName, Notification, Varbinds) -> {ok, ManagerEngineID} | {error, Reason}`

`discovery(TargetName, Notification, DiscoHandler) -> {ok, ManagerEngineID} | {error, Reason}`

`discovery(TargetName, Notification, ContextName, Varbinds) -> {ok, ManagerEngineID} | {error, Reason}`

`discovery(TargetName, Notification, Varbinds, DiscoHandler) -> {ok, ManagerEngineID} | {error, Reason}`

`discovery(TargetName, Notification, ContextName, Varbinds, DiscoHandler) -> {ok, ManagerEngineID} | {error, Reason}`

`discovery(TargetName, Notification, ContextName, Varbinds, DiscoHandler, ExtraInfo) -> {ok, ManagerEngineID} | {error, Reason}`

Types:

TargetName = string()

Notification = atom()

ContextName = string() (defaults to "")

Varbinds = varbinds()

varbinds() = [varbind()]

DiscoHandler = snmpa_discovery_handler()

ExtraInfo = term()

snmpa_discovery_handler() = Module implementing the snmpa_discovery_handler behaviour

ManagerEngineID = string()

varbind() = {Variable, Value} | {Column, RowIndex, Value} | {OID, Value}

Variable = atom()

Column = atom()

OID = oid()

Value = term()

RowIndex = [int()]

Reason = term()

Initiate the discovery process with the manager identified by `TargetName` using the notification `Notification`.

This function is synchronous, which means that it will return when the discovery process has been completed or failed.

The `DiscoHandler` module is used during the discovery process. See *discovery handler* for more info.

The `ExtraInfo` argument is passed on to the callback functions of the `DiscoHandler`.

> Note:
>
> If we are not at security-level `noAuthNoPriv`, this could be complicated, since the agent will then continue with stage 2, before which the usm-related updates must be done.

> Note:
>
> The default discovery handler will require additional actions by the caller and the discovery will not work if the security-level is higher then `noAuthNoPriv`.

`convert_config(OldConfig) -> AgentConfig`

Types:

OldConfig = list()

AgentConfig = list()

This off-line utility function can be used to convert the old snmp application config (pre snmp-4.0) to the new snmp agent config (as of snmp-4.0).

For information about the old config (`OldConfig`) see the OTP R9C documentation.

For information about the current agent config (`AgentConfig`), see either the *SNMP application* part of the reference manual or the *Configuring the application* chapter of the SNMP user's guide.

`restart_worker() -> void()`

`restart_worker(Agent) -> void()`

Types:

Agent = pid() | atom()

Restart the worker process of a multi-threaded agent.

This is a utility function, that can be useful when e.g. debugging instrumentation functions.

`restart_set_worker() -> void()`

`restart_set_worker(Agent) -> void()`

Types:

Agent = pid() | atom()

Restart the set worker process of a multi-threaded agent.

This is a utility function, that can be useful when e.g. debugging instrumentation functions.

`print_mib_info() -> void()`

Prints the content of all the (snmp) tables and variables for all mibs handled by the snmp agent.

`print_mib_tables() -> void()`

Prints the content of all the (snmp) tables for all mibs handled by the snmp agent.

`print_mib_variables() -> void()`

Prints the content of all the (snmp) variables for all mibs handled by the snmp agent.

`verbosity(Ref,Verbosity) -> void()`

Types:

> **Ref = pid() | sub_agents | master_agent | net_if | mib_server | symbolic_store | note_store | local_db**
>
> **Verbosity = verbosity() | {subagents, verbosity()}**
>
> **verbosity() = silence | info | log | debug | trace**

Sets verbosity for the designated process. For the lowest verbosity `silence`, nothing is printed. The higher the verbosity, the more is printed.

See Also

calendar(3), erlc(1)

snmpa_conf

Erlang module

The module `snmpa_conf` contains various utility functions to used for manipulating (write/append/read) the config files of the SNMP agent.

Exports

`agent_entry(Tag, Val) -> agent_entry()`

Types:

Tag = intAgentIpAddress | intAgentUDPPort | intAgentMaxPacketSize | snmpEngineMaxMessageSize | snmpEngineID

Val = term()

agent_entry() = term()

Create an entry for the agent config file, `agent.conf`.

The type of `Val` depends on the value of `Tag`, see *Agent Information* for more info.

`write_agent_config(Dir, Conf) -> ok`

`write_agent_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [agent_entry()]

Write the agent config to the agent config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Agent Information* for more info.

`append_agent_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [agent_entry()]

Append the config to the current agent config file.

`Dir` is the path to the directory where to store the config file.

See *Agent Information* for more info.

`read_agent_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [agent_entry()]

Read the current agent config file.

`Dir` is the path to the directory where to store the config file.

See *Agent Information* for more info.

`standard_entry(Tag, Val) -> standard_entry()`

Types:

Tag = sysDescr | sysObjectID | sysContact | sysName | sysLocation | sysServices | snmpEnableAuthenTraps

Val = term()

standard_entry() = term()

Create an entry for the agent standard config file, `standard.conf`.

The type of `Val` depends on the value of `Tag`, see *System Information* for more info.

`write_standard_config(Dir, Conf) -> ok`

`write_standard_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [standard_entry()]

Write the agent standard config to the agent standard config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *System Information* for more info.

`append_standard_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [standard_entry()]

Append the standard config to the current agent standard config file.

`Dir` is the path to the directory where to store the config file.

See *System Information* for more info.

`read_standard_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [standard_entry()]

Read the current agent standard config file.

`Dir` is the path to the directory where to store the config file.

See *System Information* for more info.

`context_entry(Context) -> context_entry()`

Types:

Context = string()

context_entry() = term()

Create an entry for the agent context config file, `context.conf`.

See *Contexts* for more info.

`write_context_config(Dir, Conf) -> ok`

`write_context_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [context_entry()]

Write the agent context config to the agent context config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Contexts* for more info.

`append_context_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [context_entry()]

Append the context config to the current agent context config file.

`Dir` is the path to the directory where to store the config file.

See *Contexts* for more info.

`read_context_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [context_entry()]

Read the current agent context config file.

`Dir` is the path to the directory where to store the config file.

See *Contexts* for more info.

`community_entry(CommunityIndex) -> community_entry()`

`community_entry(CommunityIndex, CommunityName, SecName, ContextName, TransportTag) -> community_entry()`

Types:

CommunityIndex = string()

CommunityName = string()

SecName = string()

CtxName = string()

TransportTag = string()

community_entry() = term()

Create an entry for the agent community config file, `community.conf`.

`CommunityIndex` must be a *non-empty* string.

`community_entry("public")` translates to the following call: `community_entry(CommunityIndex, CommunityIndex, "initial", "", "")`.

`community_entry("all-rights")` translates to the following call: `community_entry(CommunityIndex, CommunityIndex, CommunityIndex, "", "")`.

See *Community* for more info.

`write_community_config(Dir, Conf) -> ok`

`write_community_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [community_entry()]

Write the agent community config to the agent community config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Community* for more info.

`append_community_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [community_entry()]

Append the community config to the current agent community config file.

`Dir` is the path to the directory where to store the config file.

See *Community* for more info.

`read_community_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [community_entry()]

Read the current agent community config file.

`Dir` is the path to the directory where to store the config file.

See *Communities* for more info.

`target_addr_entry(Name, Ip, TagList, ParamsName, EngineId) -> target_addr_entry()`

`target_addr_entry(Name, Ip, TagList, ParamsName, EngineId, TMask) -> target_addr_entry()`

`target_addr_entry(Name, Ip, Udp, TagList, ParamsName, EngineId, TMask, MaxMessageSize) -> target_addr_entry()`

`target_addr_entry(Name, Ip, Udp, Timeout, RetryCount, TagList, ParamsName, EngineId, TMask, MaxMessageSize) -> target_addr_entry()`

Types:

Name = string()

Ip = string()

Udp = integer()

Timeout = integer()

RetryCount = integer()

TagList = string()

ParamsName = string()

EngineId = string()

TMask = string()

MaxMessageSize = integer()

target_addr_entry() = term()

Create an entry for the agent target_addr config file, `target_addr.conf`.

`Name` must be a *non-empty* string.

`target_addr_entry/5` translates to the following call: `target_addr_entry(Name, Ip, TagList, ParamsName, EngineId)`.

`target_addr_entry/6` translates to the following call: `target_addr_entry(Name, Ip, 162, TagList, ParamsName, EngineId, TMask, 2048)`.

`target_addr_entry/8` translates to the following call: `target_addr_entry(Name, Ip, Udp, 1500, 3, TagList, ParamsName, EngineId, TMask, MaxMessageSize)`.

See *Target Address Definitions* for more info.

`write_target_addr_config(Dir, Conf) -> ok`

`write_target_addr_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [target_addr_entry()]

Write the agent target_addr config to the agent target_addr config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Target Address Definitions* for more info.

append_target_addr_config(Dir, Conf) -> ok

Types:

Dir = string()

Conf = [target_addr_entry()]

Append the target_addr config to the current agent target_addr config file.

`Dir` is the path to the directory where to store the config file.

See *Target Address Definitions* for more info.

read_target_addr_config(Dir) -> Conf

Types:

Dir = string()

Conf = [target_addr_entry()]

Read the current agent target_addr config file.

`Dir` is the path to the directory where to store the config file.

See *Target Address Definitions* for more info.

target_params_entry(Name, Vsn) -> target_params_entry()

target_params_entry(Name, Vsn, SecName, SecLevel) -> target_params_entry()

target_params_entry(Name, MPModel, SecModel, SecName, SecLevel) -> target_params_entry()

Types:

Name = string()

Vsn = v1 | v2 | v3

MPModel = v1 | v2c | v3

SecModel = v1 | v2c | usm

SecName = string()

SecLevel = noAuthNoPriv | authNoPriv | authPriv

target_params_entry() = term()

Create an entry for the agent target_params config file, `target_params.conf`.

`Name` must be a *non-empty* string.

`Vsn` translates into `MPModel` and `SecModel` as follows:

```
\011  Vsn = v1 => MPModel = v1,  SecModel = v1
\011  Vsn = v2 => MPModel = v2c, SecModel = v2c
\011  Vsn = v3 => MPModel = v3,  SecModel = usm
```

`target_params_entry/2` translates to the following call: `target_params_entry(Name, Vsn, "initial", noAuthNoPriv)`.

`target_params_entry/4` translates to the following call: `target_params_entry(Name, MPModel, SecModel, SecName, SecLevel)` where `MPModel` and `SecModel` is mapped from `Vsn`, see above.

See *Target Parameters Definitions* for more info.

`write_target_params_config(Dir, Conf) -> ok`

`write_target_params_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [target_params_entry()]

Write the agent target_params config to the agent target_params config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Target Parameters Definitions* for more info.

`append_target_params_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [target_params_entry()]

Append the target_params config to the current agent target_params config file.

`Dir` is the path to the directory where to store the config file.

See *Target Parameters Definitions* for more info.

`read_target_params_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [target_params_entry()]

Read the current agent target_params config file.

`Dir` is the path to the directory where to store the config file.

See *Target Parameters Definitions* for more info.

`vacm_s2g_entry(SecModel, SecName, GroupName) -> vacm_s2g_entry()`

`vacm_acc_entry(GroupName, Prefix, SecModel, SecLevel, Match, ReadView, WriteView, NotifyView) -> vacm_acc_entry()`

`vacm_vtf_entry(ViewIndex, ViewSubtree) -> vacm_vtf_entry()`

`vacm_vtf_entry(ViewIndex, ViewSubtree, ViewStatus, ViewMask) -> vacm_vtf_entry()`

Types:

SecModel = v1 | v2c | usm

SecName = string()

GroupName = string()

Prefix = string()

SecLevel = noAuthNoPriv | authNoPriv | authPriv

Match = prefix | exact

ReadView = string()

WriteView = string()

NotifyView = string()

ViewIndex = integer()

ViewSubtree = [integer()]

ViewStatus = included | excluded

ViewMask = null | [zero_or_one()]

zero_or_one() = 0 | 1

vacm_s2g_entry() = term()

vacm_acc_entry() = term()

vacm_vtf_entry() = term()

Create an entry for the agent vacm config file, `vacm.conf`.

`vacm_vtf_entry/2` translates to the following call: `vacm_vtf_entry(ViewIndex, ViewSubtree, included, null)`.

See *MIB Views for VACM* for more info.

`write_vacm_config(Dir, Conf) -> ok`

`write_vacm_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [vacm_entry()]

vacm_entry() = vacm_sg2_entry() | vacm_acc_entry() | vacm_vtf_entry()

Write the agent vacm config to the agent vacm config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *MIB Views for VACM* for more info.

`append_vacm_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [vacm_entry()]

Append the vacm config to the current agent vacm config file.

`Dir` is the path to the directory where to store the config file.

See *MIB Views for VACM* for more info.

`read_vacm_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [vacm_entry()]

Read the current agent vacm config file.

`Dir` is the path to the directory where to store the config file.

See *MIB Views for VACM* for more info.

`usm_entry(EngineId) -> usm_entry()`

`usm_entry(EngineID, UserName, SecName, Clone, AuthP, AuthKeyC, OwnAuthKeyC, PrivP, PrivKeyC, OwnPrivKeyC, Public, AuthKey, PrivKey) -> usm_entry()`

Types:

EngineId = string()

UserName = string()

SecName = string()

Clone = zeroDotZero | [integer()]

AuthP = usmNoAuthProtocol | usmHMACMD5AuthProtocol, | usmHMACSHAAuthProtocol

AuthKeyC = string()

OwnAuthKeyC = string()

PrivP = usmNoPrivProtocol | usmDESPrivProtocol | usmAesCfb128Protocol

PrivKeyC = string()

OwnPrivKeyC = string()

Public = string()

AuthKey = [integer()]

PrivKey = [integer()]

usm_entry() = term()

Create an entry for the agent vacm config file, `vacm.conf`.

`usm_entry/1` translates to the following call: `usm_entry("initial", "initial", zeroDotZero, usmNoAuthProtocol, "", "", usmNoPrivProtocol, "", "", "", "", "")`.

See *Security data for USM* for more info.

`write_usm_config(Dir, Conf) -> ok`

`write_usm_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [usm_entry()]

Write the agent usm config to the agent usm config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Security data for USM* for more info.

`append_usm_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [usm_entry()]

Append the usm config to the current agent vacm config file.

`Dir` is the path to the directory where to store the config file.

See *Security data for USM* for more info.

`read_usm_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [usm_entry()]

Read the current agent usm config file.

`Dir` is the path to the directory where to store the config file.

See *Security data for USM* for more info.

`notify_entry(Name, Tag, Type) -> notify_entry()`

Types:

Name = string()

Tag = string()

Type = trap | inform

community_entry() = term()

Create an entry for the agent notify config file, `notify.conf`.

`Name` must be a *non-empty* string.

See *Notify Definitions* for more info.

`write_notify_config(Dir, Conf) -> ok`

`write_notify_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [notify_entry()]

Write the agent notify config to the agent notify config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Notify Definitions* for more info.

`append_notify_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [notify_entry()]

Append the notify config to the current agent notify config file.

`Dir` is the path to the directory where to store the config file.

See *Notify Definitions* for more info.

`read_notify_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [community_entry()]

Read the current agent notify config file.

`Dir` is the path to the directory where to store the config file.

See *Notify Definitions* for more info.

snmpa_discovery_handler

Erlang module

This module defines the behaviour of the agent discovery handler. A `snmpa_discovery_handler` compliant module must export the following functions:

- *stage1_finish/2*

The semantics of them and their exact signatures are explained below.

Exports

`stage1_finish(TargetName, ManagerEngineID, ExtraInfo) -> ignore | {ok, usm_entry() | [usm_entry()]} | {ok, usm_entry() | [usm_entry()], NewExtraInfo}`

Types:

TargetName = string()

ManagerEngineID = string()

ExtraInfo = term()

usm_entry() = tuple() compatible with usm.conf

NewExtraInfo = term()

This function is called at the end of stage 1 of the discovery process. It should return either the atom `ignore` or `{ok, usm_entry() | [usm_entry()]}`. See *usm_entry()* and *usm_entry/13* for more info.

If the function returns `ignore`, then it is assumed that either:

- The caller (of the discovery function) will make the needed updates later.
- The callback function itself did the updates.

In either case, the agent will do nothing, but return the retrieved ManagerEngineID (see *discovery* for more info) and possible continue with stage 2 of the discovery process.

The `ExtraInfo` argument is passed on from the *discovery* function.

This function may return an updated `NewExtraInfo` that will be used in subsequent calls to the callback functions. Intended for future use.

The purpose of this function is to generate the usm- related security data needed for usm processing in the agent. Specifically, updating the usmUserTable.

When an `usm_entry()` tuple (or a list of such tuples) is returned, this data is then added to the `usmUserTable` by the (master-) agent.

When an `usm_entry()` tuple (or a list of such tuples) is returned, this data is then added to the `usmUserTable` by the (master-) agent.

Note:

Note that the function does not check if this entry already exists.

Note:

Note that this function is executed in the context of the master-agent process.

snmpa_error_report

Erlang module

This module defines the behaviour of the agent error reporting. A `snmpa_error_report` compliant module must export the following functions:

- config_err/2
- user_err/2

The semantics of them and their exact signatures are explained below.

Exports

`config_err(Format, Args) -> void()`

Types:

Format = string()

Args = list()

The function is called if an error occurs during the configuration phase, for example if a syntax error is found in a configuration file.

`Format` and `Args` are as in `io:format(Format, Args)`.

`user_err(Format, Args) -> void()`

Types:

Format = string()

Args = list()

The function is called if a user related error occurs at run-time, for example if a user defined instrumentation function returns erroneous.

`Format` and `Args` are as in `io:format(Format, Args)`.

snmpa_error

Erlang module

The module `snmpa_error` contains two callback functions which are called if an error occurs at different times during agent operation. These functions in turn calls the corresponding function in the configured error report module, which implements the actual report functionality.

Two simple implementation(s) is provided with the toolkit; the modules *snmpa_error_logger* which is the default and *snmpa_error_io*.

The error report module is configured using the directive `error_report_mod`, see *configuration parameters*.

Exports

`config_err(Format, Args) -> void()`

Types:

Format = string()

Args = list()

The function is called if an error occurs during the configuration phase, for example if a syntax error is found in a configuration file.

`Format` and `Args` are as in `io:format(Format, Args)`.

`user_err(Format, Args) -> void()`

Types:

Format = string()

Args = list()

The function is called if a user related error occurs at run-time, for example if a user defined instrumentation function returns erroneous.

`Format` and `Args` are as in `io:format(Format, Args)`.

snmpa_error_io

Erlang module

The module `snmpa_error_io` implements the `snmp_error_report` behaviour (see *snmpa_error_report*) containing two callback functions which are called in order to report SNMP errors.

This module provides a simple mechanism for reporting SNMP errors. Errors are written to stdout using the `io` module. It is provided as an simple example.

This module needs to be explicitly configured, see *snmpa_error* and *configuration parameters*.

Exports

`config_err(Format, Args) -> void()`

Types:

Format = string()

Args = list()

The function is called if an error occurs during the configuration phase, for example if a syntax error is found in a configuration file.

`Format` and `Args` are as in `io:format(Format, Args)`.

`user_err(Format, Args) -> void()`

Types:

Format = string()

Args = list()

The function is called if a user related error occurs at run-time, for example if a user defined instrumentation function returns erroneous.

`Format` and `Args` are as in `io:format(Format, Args)`.

snmpa_error_logger

Erlang module

The module `snmpa_error_logger` implements the `snmpa_error_report` behaviour (see *snmpa_error_report*) containing two callback functions which are called in order to report SNMP errors.

This module provides a simple mechanism for reporting SNMP errors. Errors are sent to the `error_logger` after a size check. Messages are truncated after 1024 chars. It is provided as an example.

This module is the default error report module, but can be explicitly configured, see *snmpa_error* and *configuration parameters*.

Exports

`config_err(Format, Args) -> void()`

Types:

Format = string()

Args = list()

The function is called if an error occurs during the configuration phase, for example if a syntax error is found in a configuration file.

`Format` and `Args` are as in `io:format(Format, Args)`.

`user_err(Format, Args) -> void()`

Types:

Format = string()

Args = list()

The function is called if a user related error occurs at run-time, for example if a user defined instrumentation function returns erroneous.

`Format` and `Args` are as in `io:format(Format, Args)`.

See Also

error_logger(3)

snmpa_local_db

Erlang module

The module `snmpa_local_db` contains functions for implementing tables (and variables) using the SNMP built-in database. The database exists in two instances, one volatile and one persistent. The volatile database is implemented with ets. The persistent database is implemented with dets.

There is a scaling problem with this database.

- Insertions and deletions are inefficient for large tables.

This problem is best solved by using Mnesia instead.

The following functions describe the interface to `snmpa_local_db`. Each function has a Mnesia equivalent. The argument `NameDb` is a tuple `{Name, Db}` where `Name` is the symbolic name of the managed object (as defined in the MIB), and `Db` is either `volatile` or `persistent`. `mnesia` is not possible since all these functions are `snmpa_local_db` specific.

Common Data Types

In the functions defined below, the following types are used:

- `NameDb = {Name, Db}`
- `Name = atom(), Db = volatile | persistent`
- `RowIndex = [int()]`
- `Cols = [Col] | [{Col, Value}], Col = int(), Value = term()`

where `RowIndex` denotes the last part of the OID, that specifies the index of the row in the table. `Cols` is a list of column numbers in case of a get operation, and a list of column numbers and values in case of a set operation.

Exports

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

Types:

Reason = term()

This function can be used to manually dump the database to file.

`match(NameDb, Pattern)`

Performs an ets/dets matching on the table. See Stdlib documentation, module ets, for a description of `Pattern` and the return values.

`print()`

`print(TableName)`

`print(TableName, Db)`

Types:

TableName = atom()

Prints the contents of the database on screen. This is useful for debugging since the `STANDARD-MIB` and `OTP-SNMPEA-MIB` (and maybe your own MIBs) are stored in `snmpa_local_db`.

`TableName` is an atom for a table in the database. When no name is supplied, the whole database is shown.

`table_create(NameDb) -> bool()`

Creates a table. If the table already exist, the old copy is destroyed.

Returns `false` if the `NameDb` argument is incorrectly specified, `true` otherwise.

`table_create_row(NameDb, RowIndex, Row) -> bool()`

Types:

Row = {Val1, Val2, ..., ValN}
Val1 = Val2 = ... = ValN = term()

Creates a row in a table. `Row` is a tuple with values for all columns, including the index columns.

`table_delete(NameDb) -> void()`

Deletes a table.

`table_delete_row(NameDb, RowIndex) -> bool()`

Deletes the row in the table.

`table_exists(NameDb) -> bool()`

Checks if a table exists.

`table_get_row(NameDb, RowIndex) -> Row | undefined`

Types:

Row = {Val1, Val2, ..., ValN}
Val1 = Val2 = ... = ValN = term()

`Row` is a tuple with values for all columns, including the index columns.

See Also

ets(3), dets(3), snmp_generic(3)

snmpa_mpd

Erlang module

The module `snmpa_mpd` implements the version independent Message Processing and Dispatch functionality in SNMP for the agent. It is supposed to be used from a Network Interface process (*Definition of Agent Net if*).

Exports

`init(Vsns) -> mpd_state()`

Types:

Vsns = [Vsn]

Vsn = v1 | v2 | v3

This function can be called from the net_if process at start-up. The options list defines which versions to use.

It also initializes some SNMP counters.

`process_packet(Packet, TDomain, TAddress, State, NoteStore, Log) -> {ok, Vsn, Pdu, PduMS, ACMData} | {discarded, Reason} | {discovery, DiscoPacket}`

`process_packet(Packet, TDomain, TAddress, LocalEngineID, State, NoteStore, Log) -> {ok, Vsn, Pdu, PduMS, ACMData} | {discarded, Reason} | {discovery, DiscoPacket}`

Types:

Packet = binary()

TDomain = snmpUDPDomain

TAddress = {Ip, Udp}

LocalEngineID = string()

Ip = {integer(), integer(), integer(), integer()}

Udp = integer()

State = mpd_state()

NoteStore = pid()

Log = snmp_log()

Vsn = 'version-1' | 'version-2' | 'version-3'

Pdu = #pdu

PduMs = integer()

ACMData = acm_data()

Reason = term()

DiscoPacket = binary()

Processes an incoming packet. Performs authentication and decryption as necessary. The return values should be passed the agent.

> Note:
>
> Note that the use of the LocalEngineID argument is only intended for special cases, if the agent is to "emulate" multiple EngineIDs! By default, the agent uses the value of `SnmpEngineID` (see SNMP-FRAMEWORK-MIB).

generate_response_msg(Vsn, RePdu, Type, ACMData, Log) -> {ok, Packet} | {discarded, Reason}

generate_response_msg(Vsn, RePdu, Type, ACMData, LocalEngineID, Log) -> {ok, Packet} | {discarded, Reason}

Types:

- **Vsn = 'version-1' | 'version-2' | 'version-3'**
- **RePdu = #pdu**
- **Type = atom()**
- **ACMData = acm_data()**
- **LocalEngineID = string()**
- **Packet = binary()**

Generates a possibly encrypted response packet to be sent to the network. `Type` is the `#pdu.type` of the original request.

> Note:
>
> Note that the use of the LocalEngineID argument is only intended for special cases, if the agent is to "emulate" multiple EngineIDs! By default, the agent uses the value of `SnmpEngineID` (see SNMP-FRAMEWORK-MIB).

generate_msg(Vsn, NoteStore, Pdu, MsgData, To) -> {ok, PacketsAndAddresses} | {discarded, Reason}

generate_msg(Vsn, NoteStore, Pdu, MsgData, LocalEngineID, To) -> {ok, PacketsAndAddresses} | {discarded, Reason}

Types:

- **Vsn = 'version-1' | 'version-2' | 'version-3'**
- **NoteStore = pid()**
- **Pdu = #pdu**
- **MsgData = msg_data()**
- **LocalEngineID = string()**
- **To = [dest_addrs()]**
- **PacketsAndAddresses = [{TDomain, TAddress, Packet}]**
- **TDomain = snmpUDPDomain**
- **TAddress = {Ip, Udp}**
- **Ip = {integer(), integer(), integer(), integer()}**
- **Udp = integer()**
- **Packet = binary()**

Generates a possibly encrypted request packet to be sent to the network.

`MsgData` is the message specific data used in the SNMP message. This value is received in a `send_pdu` or `send_pdu_req` message from the agent. In SNMPv1 and SNMPv2c, this message data is the community string. In SNMPv3, it is the context information. `To` is a list of the destination addresses and their corresponding security parameters. This value is also received from the requests mentioned above.

Note:

Note that the use of the LocalEngineID argument is only intended for special cases, if the agent is to "emulate" multiple EngineIDs! By default, the agent uses the value of `SnmpEngineID` (see SNMP-FRAMEWORK-MIB).

`discarded_pdu(Variable) -> void()`

Types:

Variable = atom()

Increments the variable associated with a discarded pdu. This function can be used when the net_if process receives a `discarded_pdu` message from the agent.

snmpa_network_interface

Erlang module

This module defines the behaviour of the agent network interface. A `snmpa_network_interface` compliant module must export the following functions:

- *start_link/4*
- *info/1*
- *get_log_type/1*
- *set_log_type/2*
- *verbosity/2*

The semantics of them and their exact signatures are explained below.

But this is not enough. There is also a set of *mandatory* messages which the network interface entity must be able to receive and be able to send. This is described in chapter *snmp_agent_netif*.

Exports

`start_link(Prio, NoteStore, MasterAgent, Opts) -> {ok, Pid} | {error, Reason}`

Types:

Prio = priority()

NoteStore = pid()

MasterAgent = pid()

Opts = [opt()]

opt() = {verbosity, verbosity()} | {versions, versions()} | term()

versions() = [version()]

version() = v1 | v2 | v3

Start-link the network interface process.

`NoteStore` is the pid of the note-store process and `MasterAgent` is the pid of the master-agent process.

`Opts` is an (basically) implementation dependent list of options to the network interface process. There are however a number of options which *must* be handled: `versions` and `verbosity`.

`info(Pid) -> [{Key, Value}]`

Types:

Pid = pid()

The info returned is basically up to the implementer to decide. This implementation provided by the application provides info about memory allocation and various socket information.

The info returned by this function is returned together with other info collected by the agent when the *info* function is called (tagged with with the key `net_if`).

`verbosity(Pid, Verbosity) -> void()`

Types:

Pid = pid()

Verbosity = verbosity()

Change the verbosity of a running network interface process.

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

Types:

Pid = pid()

LogType = atl_type()

Reason = term()

The Audit Trail Log is managed by the network interface process. So, it is this process that has to retrieve the actual log-type.

`set_log_type(Pid, NewType) -> {ok, OldType} | {error, Reason}`

Types:

Pid = pid()

NewType = OldType = atl_type()

Reason = term()

The Audit Trail Log is managed by the network interface process. So, it is this process that has to do the actual changing of the type.

See *set_log_type* for more info.

snmpa_network_interface_filter

Erlang module

This module defines the behaviour of the agent network interface filter. A `snmpa_network_interface_filter` compliant module must export the following functions:

- *accept_recv/2*
- *accept_send/2*
- *accept_recv_pdu/3*
- *accept_send_pdu/2*

The semantics of them and their exact signatures are explained below.

The purpose of the network interface filter is to allow for filtering of messages (accept or reject) receive and send. This is done on two levels:

- The first level is at the UDP entry / exit point, i.e. immediately after the receipt of the message, before any message processing is done (accept_recv) and immediately before sending the message, after all message processing is done (accept_send).
- The second level is at the MPD entry / exit point, i.e. immediately after the basic message processing (accept_recv_pdu) / immediately before the basic message processing (accept_send_pdu).

Note that the network interface filter is something which is used by the network interface implementation provided by the application (`snmpa_net_if`). The default filter accepts all messages.

A network interface filter can e.g. be used during testing or for load regulation. If the intended use is load regulation, see also *req_limit* and the function *register_notification_filter*.

DATA TYPES

```
port() = integer() > 0
pdu_type() = 'get-request' | 'get-next-request' | 'get-response' | 'set-request' | trap | 'get-bulk-request' |
```

Exports

`accept_recv(Ip, Port) -> boolean()`

Types:

Ip = ip_address()

Port = port()

Called at the reception of a message (before *any* processing has been done).

For the message to be discarded, the function *must* return *false*.

`accept_send(Ip, Port) -> boolean()`

Types:

Ip = ip_address()

Port = port()

Called before the sending of a message (after *all* processing has been done).

For the message to be discarded, the function *must* return *false*.

`accept_recv_pdu(Ip, Port, PduType) -> boolean()`

Types:

Ip = ip_address()

Port = port()

PduType = pdu_type()

Called after the basic message processing (MPD) has been done, but before the pdu is handed over to the master-agent for primary processing.

For the pdu to be discarded, the function *must* return *false*.

`accept_send_pdu(Targets, PduType) -> Reply`

Types:

Targets = targets()

targets() = [target()]

target() = {ip_address(), port()}

PduType = pdu_type() > 0

Reply = boolean() | NewTargets

NewTargets = targets()

Called before the basic message processing (MPD) is done, when a pdu has been received from the master-agent.

For the message to be discarded all together, the function *must* return *false*.

Note that it is possible for this function to filter out targets (but *not* add its own) by returning an updated `Targets` list (`NewTargets`).

snmpa_notification_delivery_info_receiver

Erlang module

This module defines the behaviour of the notification delivery information receiver.

A `snmpa_notification_delivery_info_receiver` compliant module must export the following functions:

- *delivery_targets/3*
- *delivery_info/4*

The semantics of them and their exact signatures are explained below.

DATA TYPES

```
address() = A 4-tuple
```

Exports

`delivery_targets(Tag, Targets, Extra) -> void()`

Types:

Tag = term()

Targets = [target()]

target() = {Address, Port}

Address = address()

Port = integer()

Extra = term()

Inform about target addresses.

This is the first function called when a notification delivery is in progress. It informs the `receiver` which targets will get the notification. The result of the delivery will be provided via successive calls to `delivery_info/4` function, see below.

`delivery_info(Tag, Target, DeliveryResult, Extra) -> void()`

Types:

Tag = term()

Target = target()

target() = {Address, Port}

Address = address()

Port = integer()

DeliveryResult = delivery_result()

delivery_result() = no_response | got_response

Extra = term()

Inform about delivery result.

This function is called for each target in the Targets argument of the delivery_targets/3 function, see above.

The purpose is to inform the receiver of the result of the delivery (was the notification acknowledged or not) for each target.

snmpa_notification_filter

Erlang module

This module defines the behaviour of the agent notification filters. A `snmpa_notification_filter` compliant module must export the following functions:

- handle_notification/2

The semantics of them and their exact signatures are explained below.

The purpose of notification filters is to allow for modification and/or suppression of a notification.

A misbehaving filter will be removed.

Exports

`handle_notification(Notif, Data) -> Reply`

Types:

Reply = send | {send, NewNotif} | dont_send

Notif = NewNotif = notification() | trap()

Data = term()

Handle a notification to be sent. The filter can either accept the notification as is, return `send`, modify the notification, return `{send, NewNotif}` or suppress the notification, return `dont_send`.

`Data` is supplied at filter registration time, see *register_notification_filter*.

snmpa_supervisor

Erlang module

This is the top supervisor for the agent part of the SNMP application. There is always one supervisor at each node with an SNMP agent (master agent or sub-agent).

Exports

`start_sub_sup(Opts) -> {ok, pid()} | {error, {already_started, pid()}} | {error, Reason}`

Types:

Opts = [opt()]

opt() = {db_dir, string()} | ...

Starts a supervisor for the SNMP agent system without a master agent. The supervisor starts all involved SNMP processes, but no agent processes. Sub-agents should be started by calling `start_sub_agent/3`.

`db_dir` is mandatory.

See *configuration parameters* for a description of the options.

`start_master_sup(Opts) -> {ok, pid()} | {error, {already_started, pid()}} | {error, Reason}`

Types:

Opts = [opt()]

opt() = {db_dir, string()} | {config, ConfOpts()} | ...

ConfOpts = [conf_opts()]

conf_opts() = {dir, string()} | ...

Reason = term()

Starts a supervisor for the SNMP agent system. The supervisor starts all involved SNMP processes, including the master agent. Sub-agents should be started by calling `start_subagent/3`.

`db_dir` is mandatory.

`dir` in config is mandatory.

See *snmp config* for a description of the options.

`start_sub_agent(ParentAgent,Subtree,Mibs) -> {ok, pid()} | {error, Reason}`

Types:

ParentAgent = pid()

SubTree = oid()

Mibs = [MibName]

MibName = [string()]

Starts a sub-agent on the node where the function is called. The `snmpa_supervisor` must be running.

If the supervisor is not running, the function fails with the reason `badarg`.

`stop_sub_agent(SubAgent) -> ok | no_such_child`

Types:

SubAgent = pid()

Stops the sub-agent on the node where the function is called. The `snmpa_supervisor` must be running.

If the supervisor is not running, the function fails with the reason `badarg`.

snmp_community_mib

Erlang module

The module `snmp_community_mib` implements the instrumentation functions for the SNMP-COMMUNITY-MIB, and functions for configuring the database.

The configuration files are described in the SNMP User's Manual.

Exports

`configure(ConfDir) -> void()`

Types:

ConfDir = string()

This function is called from the supervisor at system start-up.

Inserts all data in the configuration files into the database and destroys all old rows with StorageType `volatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error, report module and the function fails with reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `community.conf`.

`reconfigure(ConfDir) -> void()`

Types:

ConfDir = string()

Inserts all data in the configuration files into the database and destroys all old data, including the rows with StorageType `nonVolatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

Thus, the data in the SNMP-COMMUNITY-MIB, after this function has been called, is from the configuration files.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `community.conf`.

`add_community(Idx, CommName, SecName, CtxName, TransportTag) -> Ret`

Types:

Idx = string()

CommName = string()

SecName = string()

CtxName = string()

TransportTag = string()

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a community to the agent config. Equivalent to one line in the `community.conf` file.

`delete_community(Key) -> Ret`

Types:

Key = term()

Ret = ok | {error, Reason}

Reason = term()

Delete a community from the agent config.

snmp_framework_mib

Erlang module

The module `snmp_framework_mib` implements instrumentation functions for the SNMP-FRAMEWORK-MIB, and functions for initializing and configuring the database.

The configuration files are described in the SNMP User's Manual.

Exports

`configure(ConfDir) -> void()`

Types:

ConfDir = string()

This function is called from the supervisor at system start-up.

Inserts all data in the configuration files into the database and destroys all old data.

Thus, the data in the SNMP-FRAMEWORK-MIB, after this function has been called, is from the configuration files.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `context.conf`.

`init() -> void()`

This function is called from the supervisor at system start-up.

Creates the necessary objects in the database if they do not exist. It does not destroy any old values.

`add_context(Ctx) -> Ret`

Types:

Ctx = string()

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a context to the agent config. Equivalent to one line in the `context.conf` file.

`delete_context(Key) -> Ret`

Types:

Key = term()

Ret = ok | {error, Reason}

Reason = term()

Delete a context from the agent config.

snmp_generic

Erlang module

The module `snmp_generic` contains generic functions for implementing tables (and variables) using the SNMP built-in database or Mnesia. These default functions are used if no instrumentation function is provided for a managed object in a MIB. Sometimes, it might be necessary to customize the behaviour of the default functions. For example, in some situations a trap should be sent if a row is deleted or modified, or some hardware is to be informed, when information is changed.

The overall structure is shown in the following figure:

```
         +---------------+
         |   SNMP Agent  |
         +- - - - - - - -+
         |      MIB      |
         +---------------+
                 |
         Association file       (associates a MIB object with
                 |               snmp_generic:table_funct
                 |               snmp_generic:variable_func)
+---------------------------------------+
|           snmp_generic                |  Support for get-next,
|                                       |  RowStatus operations
+----------------------+----------------+
|    snmpa_local_db    |    Mnesia      |  Database
+--------------+-------+----------------+
|     dets     |  ets  |
| (persistent) |       |
+--------------+-------+
```

Each function takes the argument `NameDb`, which is a tuple `{Name, Db}`, to identify which database the functions should use. `Name` is the symbolic name of the managed object as defined in the MIB, and `Db` is either `volatile`, `persistent`, or `mnesia`. If it is `mnesia`, all variables are stored in the Mnesia table `snmp_variables` which must be a table with two attributes (not a Mnesia SNMP table). The SNMP tables are stored in Mnesia tables with the same names as the SNMP tables. All functions assume that a Mnesia table exists with the correct name and attributes. It is the programmer's responsibility to ensure this. Specifically, if variables are stored in Mnesia, the table `snmp_variables` must be created by the programmer. The record definition for this table is defined in the file `snmp/include/snmp_types.hrl`.

If an instrumentation function in the association file for a variable `myVar` does not have a name when compiling an MIB, the compiler generates an entry.

```
{myVar, {snmp_generic, variable_func, [{myVar, Db]}}.
```

And for a table:

```
{myTable, {snmp_generic, table_func, [{myTable, Db]}}.
```

DATA TYPES

In the functions defined below, the following types are used:

```
name_db() = {name(), db()}
name() = atom()
db() = volatile | persistent | mnesia
row_index() = [int()]
columns() = [column()] | [{column(), value()}]
column() = int()
value() = term()
```

`row_index()`

Denotes the last part of the OID which specifies the index of the row in the table (see RFC1212, 4.1.6 for more information about INDEX).

`columns()`

Is a list of column numbers in the case of a `get` operation, and a list of column numbers and values in the case of a `set` operation.

Exports

`get_status_col(Name, Cols)`

`get_status_col(NameDb, Cols) -> {ok, StatusVal} | false`

Types:

Name = name()

NameDb = name_db()

Cols = columns()

StatusVal = term()

Gets the value of the status column from `Cols`.

This function can be used in instrumentation functions for `is_set_ok`, `undo` or `set` to check if the status column of a table is modified.

`get_index_types(Name)`

Types:

Name = name()

Gets the index types of `Name`

This function can be used in instrumentation functions to retrieve the index types part of the table info.

`table_func(Op1, NameDb)`

`table_func(Op2, RowIndex, Cols, NameDb) -> Ret`

Types:

Op1 = new | delete

Op2 = get | next | is_set_ok | set | undo

NameDb = name_db()

RowIndex = row_index()

Cols = columns()

Ret = term()

This is the default instrumentation function for tables.

- The `new` function creates the table if it does not exist, but only if the database is the SNMP internal db.
- The `delete` function does not delete the table from the database since unloading an MIB does not necessarily mean that the table should be destroyed.
- The `is_set_ok` function checks that a row which is to be modified or deleted exists, and that a row which is to be created does not exist.
- The `undo` function does nothing.
- The `set` function checks if it has enough information to make the row change its status from `notReady` to `notInService` (when a row has been set to `createAndWait`). If a row is set to `createAndWait`, columns without a value are set to `noinit`. If Mnesia is used, the set functionality is handled within a transaction.

If it is possible for a manager to create or delete rows in the table, there must be a `RowStatus` column for `is_set_ok`, `set` and `undo` to work properly.

The function returns according to the specification of an instrumentation function.

`table_get_elements(NameDb, RowIndex, Cols) -> Values`

Types:

NameDb = name_db()

RowIndex = row_index()

Cols = columns()

Values = [value() | noinit]

Returns a list with values for all columns in `Cols`. If a column is undefined, its value is `noinit`.

`table_next(NameDb, RestOid) -> RowIndex | endOfTable`

Types:

NameDb = name_db()

RestOid = [int()]

RowIndex = row_index()

Finds the indices of the next row in the table. `RestOid` does not have to specify an existing row.

`table_row_exists(NameDb, RowIndex) -> bool()`

Types:

NameDb = name_db()

RowIndex = row_index()

Checks if a row in a table exists.

`table_set_elements(NameDb, RowIndex, Cols) -> bool()`

Types:

NameDb = name_db()

RowIndex = row_index()

Cols = columns()

Sets the elements in `Cols` to the row specified by `RowIndex`. No checks are performed on the new values.

If the Mnesia database is used, this function calls `mnesia:write` to store the values. This means that this function must be called from within a transaction (`mnesia:transaction/1` or `mnesia:dirty/1`).

`variable_func(Op1, NameDb)`

`variable_func(Op2, Val, NameDb) -> Ret`

Types:

Op1 = new | delete | get

Op2 = is_set_ok | set | undo

NameDb = name_db()

Val = value()

Ret = term()

This is the default instrumentation function for variables.

The `new` function creates a new variable in the database with a default value as defined in the MIB, or a zero value (depending on the type).

The `delete` function does not delete the variable from the database.

The function returns according to the specification of an instrumentation function.

`variable_get(NameDb) -> {value, Value} | undefined`

Types:

NameDb = name_db()

Value = value()

Gets the value of a variable.

`variable_set(NameDb, NewVal) -> true | false`

Types:

NameDb = name_db()

NewVal = value()

Sets a new value to a variable. The variable is created if it does not exist. No checks are made on the type of the new value.

Returns `false` if the `NameDb` argument is incorrectly specified, otherwise `true`.

Example

The following example shows an implementation of a table which is stored in Mnesia, but with some checks performed at set-request operations.

```
myTable_func(new, NameDb) ->   % pass unchanged
  snmp_generic:table_func(new, NameDb).

myTable_func(delete, NameDb) ->   % pass unchanged
  snmp_generic:table_func(delete, NameDb).
```

```
%% change row
myTable_func(is_set_ok, RowIndex, Cols, NameDb) ->
  case snmp_generic:table_func(is_set_ok, RowIndex,
                               Cols, NameDb) of
    {noError, 0} ->
      myApplication:is_set_ok(RowIndex, Cols);
    Err ->
      Err
  end;

myTable_func(set, RowIndex, Cols, NameDb) ->
  case snmp_generic:table_func(set, RowIndex, Cols,
                               NameDb),
    {noError, 0} ->
      % Now the row is updated, tell the application
      myApplication:update(RowIndex, Cols);
    Err ->
      Err
  end;

myTable_func(Op, RowIndex, Cols, NameDb) ->   % pass unchanged
  snmp_generic:table_func(Op, RowIndex, Cols, NameDb).
```

The `.funcs` file would look like:

```
{myTable, {myModule, myTable_func, [{myTable, mnesia}]}}.
```

snmp_index

Erlang module

The module `snmp_index` implements an Abstract Data Type (ADT) for an SNMP index structure for SNMP tables. It is implemented as an ets table of the ordered_set data-type, which means that all operations are O(log n). In the table, the key is an ASN.1 OBJECT IDENTIFIER.

This index is used to separate the implementation of the SNMP ordering from the actual implementation of the table. The SNMP ordering, that is implementation of GET NEXT, is implemented in this module.

For example, suppose there is an SNMP table, which is best implemented in Erlang as one process per SNMP table row. Suppose further that the INDEX in the SNMP table is an OCTET STRING. The index structure would be created as follows:

```
snmp_index:new(string)
```

For each new process we create, we insert an item in an `snmp_index` structure:

```
new_process(Name, SnmpIndex) ->
  Pid = start_process(),
  NewSnmpIndex =
    snmp_index:insert(SnmpIndex, Name, Pid),
  <...>
```

With this structure, we can now map an OBJECT IDENTIFIER in e.g. a GET NEXT request, to the correct process:

```
get_next_pid(Oid, SnmpIndex) ->
  {ok, {_, Pid}} = snmp_index:get_next(SnmpIndex, Oid),
  Pid.
```

Common data types

The following data types are used in the functions below:

- `index()`
- `oid() = [byte()]`
- `key_types = type_spec() | {type_spec(), type_spec(), ...}`
- `type_spec() = fix_string | string | integer`
- `key() = key_spec() | {key_spec(), key_spec(), ...}`
- `key_spec() = string() | integer()`

The `index()` type denotes an snmp index structure.

The `oid()` type is used to represent an ASN.1 OBJECT IDENTIFIER.

The `key_types()` type is used when creating the index structure, and the `key()` type is used when inserting and deleting items from the structure.

The `key_types()` type defines the types of the SNMP INDEX columns for the table. If the table has one single INDEX column, this type should be a single atom, but if the table has multiple INDEX columns, it should be a tuple with atoms.

If the INDEX column is of type INTEGER, or derived from INTEGER, the corresponding type should be `integer`. If it is a variable length type (e.g. OBJECT IDENTIFIER, OCTET STRING), the corresponding type should be `string`. Finally, if the type is of variable length, but with a fixed size restriction (e.g. IpAddress), the corresponding type should be `fix_string`.

For example, if the SNMP table has two INDEX columns, the first one an OCTET STRING with size 2, and the second one an OBJECT IDENTIFER, the corresponding `key_types` parameter would be `{fix_string, string}`.

The `key()` type correlates to the `key_types()` type. If the `key_types()` is a single atom, the corresponding `key()` is a single type as well, but if the `key_types()` is a tuple, `key` must be a tuple of the same size.

In the example above, valid `keys` could be `{"hi", "mom"}` and `{"no", "thanks"}`, whereas `"hi"`, `{"hi", 42}` and `{"hello", "there"}` would be invalid.

Warning:

All API functions that update the index return a `NewIndex` term. This is for backward compatibility with a previous implementation that used a B+ tree written purely in Erlang for the index. The `NewIndex` return value can now be ignored. The return value is now the unchanged table identifier for the ets table.

The implementation using ets tables introduces a semantic incompatibility with older implementations. In those older implementations, using pure Erlang terms, the index was garbage collected like any other Erlang term and did not have to be deleted when discarded. An ets table is deleted only when the process creating it explicitly deletes it or when the creating process terminates.

A new interface `delete/1` is now added to handle the case when a process wants to discard an index table (i.e. to build a completely new). Any application using transient snmp indexes has to be modified to handle this.

As an snmp adaption usually keeps the index for the whole of the systems lifetime, this is rarely a problem.

Exports

`delete(Index) -> true`

Types:

Index = NewIndex = index()

Key = key()

Deletes a complete index structure (i.e. the ets table holding the index). The index can no longer be referenced after this call. See the *warning note* above.

`delete(Index, Key) -> NewIndex`

Types:

Index = NewIndex = index()

Key = key()

Deletes a key and its value from the index structure. Returns a new structure.

`get(Index, KeyOid) -> {ok, {KeyOid, Value}} | undefined`

Types:

Index = index()

KeyOid = oid()

Value = term()

Gets the item with key `KeyOid`. Could be used from within an SNMP instrumentation function.

`get_last(Index) -> {ok, {KeyOid, Value}} | undefined`

Types:

Index = index()

KeyOid = oid()

Value = term()

Gets the last item in the index structure.

`get_next(Index, KeyOid) -> {ok, {NextKeyOid, Value}} | undefined`

Types:

Index = index()

KeyOid = NextKeyOid = oid()

Value = term()

Gets the next item in the SNMP lexicographic ordering, after `KeyOid` in the index structure. `KeyOid` does not have to refer to an existing item in the index.

`insert(Index, Key, Value) -> NewIndex`

Types:

Index = NewIndex = index()

Key = key()

Value = term()

Inserts a new key value tuple into the index structure. If an item with the same key already exists, the new `Value` overwrites the old value.

`key_to_oid(Index, Key) -> KeyOid`

Types:

Index = index()

Key = key()

KeyOid = NextKeyOid = oid()

Converts `Key` to an OBJECT IDENTIFIER.

`new(KeyTypes) -> Index`

Types:

KeyTypes = key_types()

Index = index()

Creates a new snmp index structure. The `key_types()` type is described above.

snmp_notification_mib

Erlang module

The module `snmp_notification_mib` implements the instrumentation functions for the SNMP-NOTIFICATION-MIB, and functions for configuring the database.

The configuration files are described in the SNMP User's Manual.

Exports

configure(ConfDir) -> void()

Types:

ConfDir = string()

This function is called from the supervisor at system start-up.

Inserts all data in the configuration files into the database and destroys all old rows with StorageType `volatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `notify.conf`.

reconfigure(ConfDir) -> void()

Types:

ConfDir = string()

Inserts all data in the configuration files into the database and destroys all old data, including the rows with StorageType `nonVolatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

Thus, the data in the SNMP-NOTIFICATION-MIB, after this function has been called, is from the configuration files.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `notify.conf`.

add_notify(Name, Tag, Type) -> Ret

Types:

Name = string()

Tag = string()

Type = trap | inform

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a notify definition to the agent config. Equivalent to one line in the `notify.conf` file.

`delete_notify(Key) -> Ret`

Types:

- **Key = term()**
- **Ret = ok | {error, Reason}**
- **Reason = term()**

Delete a notify definition from the agent config.

snmp_pdus

Erlang module

RFC1157, RFC1905 and/or RFC2272 should be studied carefully before using this module, `snmp_pdus`.

The module `snmp_pdus` contains functions for encoding and decoding of SNMP protocol data units (PDUs). In short, this module converts a list of bytes to Erlang record representations and vice versa. The record definitions can be found in the file `snmp/include/snmp_types.hrl`. If snmpv3 is used, the module that includes `snmp_types.hrl` must define the constant `SNMP_USE_V3` before the header file is included. Example:

```
-define(SNMP_USE_V3, true).
-include_lib("snmp/include/snmp_types.hrl").
```

Encoding and decoding must be done explicitly when writing your own Net if process.

Exports

dec_message([byte()]) -> Message

Types:

Message = #message

Decodes a list of bytes into an SNMP Message. Note, if there is a v3 message, the `msgSecurityParameters` are not decoded. They must be explicitly decoded by a call to a security model specific decoding function, e.g. `dec_usm_security_parameters/1`. Also note, if the `scopedPDU` is encrypted, the OCTET STRING encoded `encryptedPDU` will be present in the `data` field.

dec_message_only([byte()]) -> Message

Types:

Message = #message

Decodes a list of bytes into an SNMP Message, but does not decode the data part of the Message. That means, data is still a list of bytes, normally an encoded `PDU` (v1 and V2) or an encoded and possibly encrypted `scopedPDU` (v3).

dec_pdu([byte()]) -> Pdu

Types:

Pdu = #pdu

Decodes a list of bytes into an SNMP Pdu.

dec_scoped_pdu([byte()]) -> ScopedPdu

Types:

ScopedPdu = #scoped_pdu

Decodes a list of bytes into an SNMP ScopedPdu.

dec_scoped_pdu_data([byte()]) -> ScopedPduData

Types:

ScopedPduData = #scoped_pdu | EncryptedPDU

EncryptedPDU = [byte()]

Decodes a list of bytes into either a scoped pdu record, or - if the scoped pdu was encrypted - to a list of bytes.

dec_usm_security_parameters([byte()]) -> UsmSecParams

Types:

UsmSecParams = #usmSecurityParameters

Decodes a list of bytes into an SNMP UsmSecurityParameters

enc_encrypted_scoped_pdu(EncryptedScopedPdu) -> [byte()]

Types:

EncryptedScopedPdu = [byte()]

Encodes an encrypted SNMP ScopedPdu into an OCTET STRING that can be used as the `data` field in a `message` record, that later can be encoded with a call to `enc_message_only/1`.

This function should be used whenever the `ScopedPDU` is encrypted.

enc_message(Message) -> [byte()]

Types:

Message = #message

Encodes a message record to a list of bytes.

enc_message_only(Message) -> [byte()]

Types:

Message = #message

`Message` is a record where the `data` field is assumed to be encoded (a list of bytes). If there is a v1 or v2 message, the `data` field is an encoded `PDU`, and if there is a v3 message, `data` is an encoded and possibly encrypted `scopedPDU`.

enc_pdu(Pd) -> [byte()]

Types:

Pdu = #pdu

Encodes an SNMP Pdu into a list of bytes.

enc_scoped_pdu(ScopedPdu) -> [byte()]

Types:

ScopedPdu = #scoped_pdu

Encodes an SNMP ScopedPdu into a list of bytes, which can be encrypted, and after encryption, encoded with a call to `enc_encrypted_scoped_pdu/1`; or it can be used as the `data` field in a `message` record, which then can be encoded with `enc_message_only/1`.

enc_usm_security_parameters(UsmSecParams) -> [byte()]

Types:

UsmSecParams = #usmSecurityParameters

Encodes SNMP UsmSecurityParameters into a list of bytes.

snmp_standard_mib

Erlang module

The module `snmp_standard_mib` implements the instrumentation functions for the STANDARD-MIB and SNMPv2-MIB, and functions for configuring the database.

The configuration files are described in the SNMP User's Manual.

Exports

configure(ConfDir) -> void()

Types:

ConfDir = string()

This function is called from the supervisor at system start-up.

Inserts all data in the configuration files into the database and destroys all old rows with StorageType `volatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with the reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `standard.conf`.

inc(Name) -> void()

inc(Name, N) -> void()

Types:

Name = atom()

N = integer()

Increments a variable in the MIB with `N`, or one if `N` is not specified.

reconfigure(ConfDir) -> void()

Types:

ConfDir = string()

Inserts all data in the configuration files into the database and destroys all old data, including the rows with StorageType `nonVolatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

Thus, the data in the SNMP-STANDARD-MIB and SNMPv2-MIB, after this function has been called, is from the configuration files.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with the reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `standard.conf`.

`reset() -> void()`

Resets all `snmp` counters to 0.

`sys_up_time() -> Time`

Types:

Time = int()

Gets the system up time in hundredth of a second.

snmp_target_mib

Erlang module

The module `snmp_target_mib` implements the instrumentation functions for the SNMP-TARGET-MIB, and functions for configuring the database.

The configuration files are described in the SNMP User's Manual.

Exports

`configure(ConfDir) -> void()`

Types:

ConfDir = string()

This function is called from the supervisor at system start-up.

Inserts all data in the configuration files into the database and destroys all old rows with StorageType `volatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with the reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration files read are: `target_addr.conf` and `target_params.conf`.

`reconfigure(ConfDir) -> void()`

Types:

ConfDir = string()

Inserts all data in the configuration files into the database and destroys all old data, including the rows with StorageType `nonVolatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

Thus, the data in the SNMP-TARGET-MIB, after this function has been called, is the data from the configuration files.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the , and the function fails with the reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration files read are: `target_addr.conf` and `target_params.conf`.

`set_target_engine_id(TargetAddrName, EngineId) -> boolean()`

Types:

TargetAddrName = string()

EngineId = string()

Changes the engine id for a target in the `snmpTargetAddrTable`. If notifications are sent as Inform requests to a target, its engine id must be set.

`add_addr(Name, Ip, Port, Timeout, Retry, TagList, Params, EngineId, TMask, MMS) -> Ret`

Types:

Name = string()

Ip = [integer()], length 4

Port = integer()

Timeout = integer()

Retry = integer()

TagList = string()

ParamsName = string()

EngineId = string()

TMask = string(), length 0 or 6

MMS = integer()

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a target address definition to the agent config. Equivalent to one line in the `target_addr.conf` file.

`delete_addr(Key) -> Ret`

Types:

Key = term()

Ret = ok | {error, Reason}

Reason = term()

Delete a target address definition from the agent config.

`add_params(Name, MPModel, SecModel, SecName, SecLevel) -> Ret`

Types:

Name = string()

MPModel = v1 | v2c | v3

SecModel = v1 | v2c | usm

SecName = string()

SecLevel = noAuthNoPriv | authNoPriv | authPriv

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a target parameter definition to the agent config. Equivalent to one line in the `target_params.conf` file.

`delete_params(Key) -> Ret`

Types:

Key = term()

Ret = ok | {error, Reason}

Reason = term()

Delete a target parameter definition from the agent config.

snmp_user_based_sm_mib

Erlang module

The module `snmp_user_based_sm_mib` implements the instrumentation functions for the SNMP-USER-BASED-SM-MIB, and functions for configuring the database.

The configuration files are described in the SNMP User's Manual.

Exports

`configure(ConfDir) -> void()`

Types:

ConfDir = string()

This function is called from the supervisor at system start-up.

Inserts all data in the configuration files into the database and destroys all old rows with StorageType `volatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with the reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `usm.conf`.

`reconfigure(ConfDir) -> void()`

Types:

ConfDir = string()

Inserts all data in the configuration files into the database and destroys all old data, including the rows with StorageType `nonVolatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

Thus, the data in the SNMP-USER-BASED-SM-MIB, after this function has been called, is the data from the configuration files.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with the reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `usm.conf`.

`add_user(EngineID, Name, SecName, Clone, AuthP, AuthKeyC, OwnAuthKeyC, PrivP, PrivKeyC, OwnPrivKeyC, Public, AuthKey, PrivKey) -> Ret`

Types:

EngineID = string()

Name = string()

SecName = string()

Clone = zeroDotZero | [integer()]

AuthP = usmNoAuthProtocol | usmHMACMD5AuthProtocol | usmHMACSHAAuthProtocol

AuthKeyC = string()

OwnAuthKeyC = string()

PrivP = usmNoPrivProtocol | usmDESPrivProtocol

PrivKeyC = string()

OwnPrivKeyC = string()

Public = string()

AuthKey = string()

PrivKey = string()

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a USM security data (user) to the agent config. Equivalent to one line in the `usm.conf` file.

`delete_user(Key) -> Ret`

Types:

Key = term()

Ret = ok | {error, Reason}

Reason = term()

Delete a USM security data (user) from the agent config.

snmp_view_based_acm_mib

Erlang module

The module `snmp_view_based_acm_mib` implements the instrumentation functions for the SNMP-VIEW-BASED-ACM-MIB, and functions for configuring the database.

The configuration files are described in the SNMP User's Manual.

Exports

`configure(ConfDir) -> void()`

Types:

ConfDir = string()

This function is called from the supervisor at system start-up.

Inserts all data in the configuration files into the database and destroys all old rows with StorageType `volatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function `config_err/2` of the error report module, and the function fails with the reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `vacm.conf`.

`reconfigure(ConfDir) -> void()`

Types:

ConfDir = string()

Inserts all data in the configuration files into the database and destroys all old data, including the rows with StorageType `nonVolatile`. The rows created from the configuration file will have StorageType `nonVolatile`.

Thus, the data in the SNMP-VIEW-BASED-ACM-MIB, after this function has been called, is the data from the configuration files.

All `snmp` counters are set to zero.

If an error is found in the configuration file, it is reported using the function *config_err/2* of the error report module, and the function fails with the reason `configuration_error`.

`ConfDir` is a string which points to the directory where the configuration files are found.

The configuration file read is: `vacm.conf`.

`add_sec2group(SecModel, SecName, GroupName) -> Ret`

Types:

SecModel = v1 | v2c | usm

SecName = string()

GroupName = string()

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a security to group definition to the agent config. Equivalent to one vacmSecurityToGroup-line in the `vacm.conf` file.

`delete_sec2group(Key) -> Ret`

Types:

Key = term()

Ret = ok | {error, Reason}

Reason = term()

Delete a security to group definition from the agent config.

`add_access(GroupName, Prefix, SecModel, SecLevel, Match, RV, WV, NV) -> Ret`

Types:

GroupName = string()

Prefix = string()

SecModel = v1 | v2c | usm

SecLevel = string()

Match = prefix | exact

RV = string()

WV = string()

NV = string()

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a access definition to the agent config. Equivalent to one vacmAccess-line in the `vacm.conf` file.

`delete_access(Key) -> Ret`

Types:

Key = term()

Ret = ok | {error, Reason}

Reason = term()

Delete a access definition from the agent config.

`add_view_tree_fam(ViewIndex, SubTree, Status, Mask) -> Ret`

Types:

ViewIndex = integer()

SubTree = oid()

Status = included | excluded

Mask = null | [integer()], where all values are either 0 or 1

Ret = {ok, Key} | {error, Reason}

Key = term()

Reason = term()

Adds a view tree family definition to the agent config. Equivalent to one vacmViewTreeFamily-line in the `vacm.conf` file.

`delete_view_tree_fam(Key) -> Ret`

Types:

Key = term()

Ret = ok | {error, Reason}

Reason = term()

Delete a view tree family definition from the agent config.

snmpc

Erlang module

The module `snmpc` contains interface functions to the SNMP toolkit MIB compiler.

Exports

`compile(File)`

`compile(File, Options) -> {ok, BinFileName} | {error, Reason}`

Types:

File = string()

Options = [opt()]

opt() = db() | relaxed_row_name_assign_check() | deprecated() | description() | reference() | group_check() | i() | il() | imports() | module() | module_identity() | module_compliance() | agent_capabilities() | outdir() | no_defs() | verbosity() | warnings()

db() = {db, volatile|persistent|mnesia}

deprecated() = {deprecated, bool()}

relaxed_row_name_assign_check() = relaxed_row_name_assign_check

description() = description

reference() = reference

group_check() = {group_check, bool()}

i() = {i, [dir()]}

il() = {il, [dir()]}

imports() = imports

module() = {module, atom()}

module_identity() = module_identity

module_compliance() = module_compliance

agent_capabilities() = agent_capabilities

no_defs() = no_defs

outdir() = {outdir, dir()}

verbosity() = {verbosity, silence|warning|info|log|debug|trace}

warnings() = {warnings, bool()}

dir() = string()

BinFileName = string()

Compiles the specified MIB file `<File>.mib`. The compiled file `BinFileName` is called `<File>.bin`.

- The option `db` specifies which database should be used for the default instrumentation.

  Default is `volatile`.
- The option `deprecated` specifies if a deprecated definition should be kept or not. If the option is false the MIB compiler will ignore all deprecated definitions.

  Default is `true`.

- The option `relaxed_row_name_assign_check`, if present, specifies that the row name assign check shall not be done strictly according to the SMI (which allows only the value 1). With this option, all values greater than zero is allowed (>= 1). This means that the error will be converted to a warning.

  By default it is not included, but if this option is present it will be.
- The option `description` specifies if the text of the DESCRIPTION field will be included or not.

  By default it is not included, but if this option is present it will be.
- The option `reference` specifies if the text of the REFERENCE field, when found in a table definition, will be included or not.

  By default it is not included, but if this option is present it will be. The reference text will be placed in the allocList field of the mib-entry record (#me{}) for the table.
- The option `group_check` specifies whether the mib compiler should check the OBJECT-GROUP macro and the NOTIFICATION-GROUP macro for correctness or not.

  Default is `true`.
- The option `i` specifies the path to search for imported (compiled) MIB files. The directories should be strings with a trailing directory delimiter.

  Default is `["./"]`.
- The option `il` (include_lib) also specifies a list of directories to search for imported MIBs. It assumes that the first element in the directory name corresponds to an OTP application. The compiler will find the current installed version. For example, the value ["snmp/mibs/"] will be replaced by ["snmp-3.1.1/mibs/"] (or what the current version may be in the system). The current directory and the `<snmp-home>/priv/mibs/` are always listed last in the include path.
- The option `imports`, if present, specifies that the IMPORT statement of the MIB shall be included in the compiled mib.
- The option `module`, if present, specifies the name of a module which implements all instrumentation functions for the MIB.

  The name of all instrumentation functions must be the same as the corresponding managed object it implements.
- The option `module_identity`, if present, specifies that the info part of the MODULE-IDENTITY statement of the MIB shall be included in the compiled mib.
- The option `module_compliance`, if present, specifies that the MODULE-COMPLIANCE statement of the MIB shall be included (with a mib-entry record) in the compiled mib. The mib-entry record of the module-compliance will contain `reference` and `module` part(s) this info in the `assocList` field).
- The option `agent_capabilities`, if present, specifies that the AGENT-CAPABILITIES statement of the MIB shall be included (with a mib-entry record) in the compiled mib. The mib-entry record of the agent-capabilitie will contain `reference` and `modules` part(s) this info in the `assocList` field).
- The option `no_defs`, if present, specifies that if a managed object does not have an instrumentation function, the default instrumentation function should NOT be used, instead this is reported as an error, and the compilation aborts.
- The option `verbosity` specifies the verbosity of the SNMP mib compiler. I.e. if warning, info, log, debug and trace messages shall be shown.

  Default is `silence`.

  Note that if the option `warnings` is `true` and the option `verbosity` is `silence`, warning messages will still be shown.
- The option `warnings` specifies whether warning messages should be shown.

  Default is `true`.

The MIB compiler understands both SMIv1 and SMIv2 MIBs. It uses the `MODULE-IDENTITY` statement to determine if the MIB is version 1 or 2.

The MIB compiler can be invoked from the OS command line by using the command `erlc`. `erlc` recognizes the extension `.mib`, and invokes the SNMP MIB compiler for files with that extension. The options `db`, `group_check`, `deprecated`, `description`, `verbosity`, `imports` and `module_identity` have to be specified to `erlc` using the syntax `+term`. See `erlc(1)` for details.

`is_consistent(Mibs) -> ok | {error, Reason}`

Types:

Mibs = [MibName]

MibName = string()

Checks for multiple usage of object identifiers and traps between MIBs.

`mib_to_hrl(MibName) -> ok | {error, Reason}`

Types:

MibName = string()

Generates a `.hrl` file with definitions of Erlang constants for the objects in the MIB. The `.hrl` file is called `<MibName>.hrl`. The MIB must be compiled, and present in the current directory.

The `mib_to_hrl` generator can be invoked from the OS command line by using the command `erlc`. `erlc` recognizes the extension `.bin`, and invokes this function for files with that extension.

See Also

erlc(1)

snmpc(command)

Command

The `snmpc` program provides a way to run the SNMP MIB compiler of the Erlang system.

Exports

`snmpc [options] file.mib | file.bin`

`snmpc` compile a SNMP MIB file, see *compile/1,2* for more info.

It can also be used to generate a header file (.hrl) with definitions of Erlang constants for the objects in the MIB, see *mib_to_hrl/1*.

Compiler options

The following options are supported (note that most of these relate to the compilation of the MIB file):

--help

Prints help info.

--version

Prints application and mib format version.

--verbosity *verbosity*

Print debug info.

`verbosity` = `trace` | `debug` | `log` | `info` | `silence`

Defaults to `silence`.

--warnings

Print warning messages.

--o *directory*

The directory where the compiler should place the output files. If not specified, output files will be placed in the current working directory.

--i *Directory*

Specifies the path to search for imported (compiled) MIB files. By default, the current working directory is always included.

This option can be present several times, each time specifying *one* path.

--il *Directory*

This option (include_lib), specifies a list of directories to search for imported MIBs. It assumes that the first element in the directory name corresponds to an OTP application. The compiler will find the current installed version. For example, the value ["snmp/mibs/"] will be replaced by ["snmp-3.1.1/mibs/"] (or what the current version may be in the system). The current directory and the "snmp-home"/priv/mibs/ are always listed last in the include path.

--sgc

This option (skip group check), if present, disables the group check of the mib compiler. That is, should the OBJECT-GROUP and the NOTIFICATION-GROUP macro(s) be checked for correctness or not.

--dep

Keep deprecated definition(s). If not specified the compiler will ignore deprecated definitions.

--desc

The DESCRIPTION field will be included.

--ref

The REFERENCE field will be included.

--imp

The IMPORTS field will be included.

--mi

The MODULE-IDENTITY field will be included.

--mc

The MODULE-COMPLIANCE field will be included.

--ac

The AGENT-CAPABILITIES field will be included.

--mod *module*

The module which implements all the instrumentation functions.

The name of all instrumentation functions must be the same as the corresponding managed object it implements.

--nd

The default instrumentation functions will *not* be used if a managed object have no instrumentation function. Instead this will be reported as an error, and the compilation aborts.

--rrnac

This option, if present, specifies that the row name assign check shall not be done strictly according to the SMI (which allows only the value 1).

With this option, all values greater than zero is allowed (>= 1). This means that the error will be converted to a warning.

By default it is not included, but if this option is present it will be.

SEE ALSO

erlc(1), compile(3), snmpc(3)

snmpm

Erlang module

The module `snmpm` contains interface functions to the SNMP manager.

Common Data Types

The following data types are used in the functions below:

```
oid() = [byte()]  -  The oid() type is used to represent an ASN.1 OBJECT IDENTIFIER
snmp_reply() = {error_status(), error_index(), varbinds()}
error_status() = noError | atom()
error_index() = integer()
varbinds() = [varbind()]
atl_type() = read | write | read_write
target_name() = string()  -  Is a unique *non-empty* string
vars_and_vals() = [var_and_val()]
var_and_val() = {oid(), value_type(), value()} | {oid(), value()}
value_type() = o ('OBJECT IDENTIFIER') |
               i ('INTEGER') |
               u ('Unsigned32') |
               g ('Unsigned32') |
               s ('OCTET SRING') |
               b ('BITS') |
               ip ('IpAddress') |
               op ('Opaque') |
               c32 ('Counter32') |
               c64 ('Counter64') |
               tt ('TimeTicks')
value() = term()
```

Exports

`monitor() -> Ref`

Types:

Ref = reference()

Monitor the SNMP manager. In case of a crash, the calling (monitoring) process will get a 'DOWN' message (see the erlang module for more info).

`demonitor(Ref) -> void()`

Types:

Ref = reference()

Turn off monitoring of the SNMP manager.

`notify_started(Timeout) -> Pid`

Types:

Timeout = integer()

Pid = pid()

Request a notification (message) when the SNMP manager has started.

The `Timeout` is the time the request is valid. The value has to be greater then zero.

The `Pid` is the process handling the supervision of the SNMP manager start. When the manager has started a completion message will be sent to the client from this process: `{snmpm_started, Pid}`. If the SNMP manager was not started in time, a timeout message will be sent to the client: `{snmpm_start_timeout, Pid}`.

A client application that is dependent on the SNMP manager will use this function in order to be notified of when the manager has started. There are two situations when this is useful:

- During the start of a system, when a client application *could* start prior to the SNMP manager but is dependent upon it, and therefor has to wait for it to start.
- When the SNMP manager has crashed, the dependent client application has to wait for the SNMP manager to be restarted before it can *reconnect*.

The function returns the pid() of a handler process, that does the supervision on behalf of the client application. Note that the client application is linked to this handler.

This function is used in conjunction with the monitor function.

cancel_notify_started(Pid) -> void()

Types:

Pid = pid()

Cancel a previous request to be notified of SNMP manager start.

register_user(Id, Module, Data) -> ok | {error, Reason}

register_user(Id, Module, Data, DefaultAgentConfig) -> ok | {error, Reason}

Types:

Id = term()

Module = snmpm_user()

Data = term()

DefaultAgentConfig = [default_agent_config()]

default_agent_config() = {Item, Val}

Item = community | timeout | max_message_size | version | sec_model | sec_name | sec_level

Val = term()

Reason = term()

snmpm_user() = Module implementing the snmpm_user behaviour

Register the manager entity (=user) responsible for specific agent(s).

`Module` is the callback module (snmpm_user behaviour) which will be called whenever something happens (detected agent, incoming reply or incoming trap/notification). Note that this could have already been done as a consequence of the node config. (see users.conf).

The argument `DefaultAgentConfig` is used as default values when this user register agents.

The type of `Val` depends on `Item`:

```
community = string()
timeout = integer() | snmp_timer()
max_message_size = integer()
version = v1 | v2 | v3
sec_model = any | v1 | v2c | usm
```

```
sec_name = string()
sec_level = noAuthNoPriv | authNoPriv | authPriv
```

register_user_monitor(Id, Module, Data) -> ok | {error, Reason}

register_user_monitor(Id, Module, Data, DefaultAgentConfig) -> ok | {error, Reason}

Types:

Id = term()

Module = snmpm_user()

DefaultAgentConfig = [default_agent_config()]

default_agent_config() = {Item, Val}

Item = community | timeout | max_message_size | version | sec_model | sec_name | sec_level

Val = term()

Data = term()

Reason = term()

snmpm_user() = Module implementing the snmpm_user behaviour

Register the monitored manager entity (=user) responsible for specific agent(s).

The process performing the registration will be monitored. Which means that if that process should die, all agents registered by that user process will be unregistered. All outstanding requests will be canceled.

`Module` is the callback module (snmpm_user behaviour) which will be called whenever something happens (detected agent, incoming reply or incoming trap/notification). Note that this could have already been done as a consequence of the node config. (see users.conf).

The argument `DefaultAgentConfig` is used as default values when this user register agents.

The type of `Val` depends on `Item`:

```
community = string()
timeout = integer() | snmp_timer()
max_message_size = integer()
version = v1 | v2 | v3
sec_model = any | v1 | v2c | usm
sec_name = string()
sec_level = noAuthNoPriv | authNoPriv | authPriv
```

unregister_user(Id) -> ok | {error, Reason}

Types:

Id = term()

Unregister the user.

which_users() -> Users

Types:

Users = [UserId]

UserId = term()

Get a list of the identities of all registered users.

`register_agent(UserId, TargetName, Config) -> ok | {error, Reason}`

Types:

UserId = term()

Addr = ip_address()

TargetName = target_name()

Config = [agent_config()]

agent_config() = {Item, Val}

Item = engine_id | address | port | community | timeout | max_message_size | version | sec_model | sec_name | sec_level

Val = term()

Reason = term()

Explicitly instruct the manager to handle this agent, with `UserId` as the responsible user.

Called to instruct the manager that this agent shall be handled. This function is used when the user knows in advance which agents the manager shall handle. Note that there is an alternate way to do the same thing: Add the agent to the manager config files (see *agents.conf*).

`TargetName` is a non-empty string, uniquely identifying the agent.

The type of `Val` depends on `Item`:

```
[mandatory] engine_id = string()
[mandatory] address = ip_address()
[optional]  port = integer()
[optional]  community = string()
[optional]  timeout = integer() | snmp_timer()
[optional]  max_message_size = integer()
[optional]  version = v1 | v2 | v3
[optional]  sec_model = any | v1 | v2c | usm
[optional]  sec_name = string()
[optional]  sec_level = noAuthNoPriv | authNoPriv | authPriv
```

Note that if no `Port` is given, the default value is used.

`unregister_agent(UserId, TargetName) -> ok | {error, Reason}`

Types:

UserId = term()

TargetName = target_name()

Unregister the agent.

`agent_info(TargetName, Item) -> {ok, Val} | {error, Reason}`

Types:

TargetName = target_name()

Item = atom()

Reason = term()

Retrieve agent config.

update_agent_info(UserId, TargetName, Item, Val) -> ok | {error, Reason}

Types:

UserId = term()

TargetName = target_name()

Item = atom()

Val = term()

Reason = term()

Update agent config.

which_agents() -> Agents

which_agents(UserId) -> Agents

Types:

UserId = term()

Agents = [TargetName]

TargetName = target_name()

Get a list of all registered agents or all agents registered by a specific user.

register_usm_user(EngineID, UserName, Conf) -> ok | {error, Reason}

Types:

EngineID = string()

UserName = string()

Conf = [usm_config()]

usm_config() = {Item, Val}

Item = sec_name | auth | auth_key | priv | priv_key

Val = term()

Reason = term()

Explicitly instruct the manager to handle this USM user. Note that there is an alternate way to do the same thing: Add the usm user to the manager config files (see *usm.conf*).

The type of `Val` depends on `Item`:

```
sec_name = string()
auth = usmNoAuthProtocol | usmHMACMD5AuthProtocol | usmHMACSHAAuthProtocoltimeout
auth_key = [integer()]   (length 16 if auth = usmHMACMD5AuthProtocol,
                          length 20 if auth = usmHMACSHAAuthProtocol)
priv = usmNoPrivProtocol | usmDESPrivProtocol | usmAesCfb128Protocol
priv_key = [integer()]   (length is 16 if priv = usmDESPrivProtocol | usmAesCfb128Protocol).
```

unregister_usm_user(EngineID, UserName) -> ok | {error, Reason}

Types:

EngineID = string()

UserName = string()

Reason = term()

Unregister this USM user.

`usm_user_info(EngineID, UserName, Item) -> {ok, Val} | {error, Reason}`

Types:

EngineID = string()

UsmName = string()

Item = sec_name | auth | auth_key | priv | priv_key

Reason = term()

Retrieve usm user config.

`update_usm_user_info(EngineID, UserName, Item, Val) -> ok | {error, Reason}`

Types:

EngineID = string()

UsmName = string()

Item = sec_name | auth | auth_key | priv | priv_key

Val = term()

Reason = term()

Update usm user config.

`which_usm_users() -> UsmUsers`

Types:

UsmUsers = [{EngineID,UserName}]

EngineID = string()

UsmName = string()

Get a list of all registered usm users.

`which_usm_users(EngineID) -> UsmUsers`

Types:

UsmUsers = [UserName]

UserName = string()

Get a list of all registered usm users with engine-id `EngineID`.

`sync_get(UserId, TargetName, Oids) -> {ok, SnmpReply, Remaining} | {error, Reason}`

`sync_get(UserId, TargetName, ContextName, Oids) -> {ok, SnmpReply, Remaining} | {error, Reason}`

`sync_get(UserId, TargetName, Oids, Timeout) -> {ok, SnmpReply, Remaining} | {error, Reason}`

`sync_get(UserId, TargetName, ContextName, Oids, Timeout) -> {ok, SnmpReply, Remaining} | {error, Reason}`

`sync_get(UserId, TargetName, ContextName, Oids, Timeout, ExtraInfo) -> {ok, SnmpReply, Remaining} | {error, Reason}`

Types:

UserId = term()

TargetName = target_name()

ContextName = string()

Oids = [oid()]

Timeout = integer()

ExtraInfo = term()

SnmpReply = snmp_reply()

Remaining = integer()

Reason = {send_failed, ReqId, R} | {invalid_sec_info, SecInfo, SnmpInfo} | term()

R = term()

SecInfo = [sec_info()]

sec_info() = {sec_tag(), ExpectedValue, ReceivedValue}

sec_tag() = atom()

ExpectedValue = ReceivedValue = term()

SnmpInfo = term()

Synchronous `get-request`.

`Remaining` is the remaining time of the given or default timeout time.

When *Reason* is *{send_failed, ...}* it means that the net_if process failed to send the message. This could happen because of any number of reasons, i.e. encoding error. *R* is the actual reason in this case.

`ExtraInfo` is an opaque data structure passed on to the net-if process. The net-if process included in this application makes no use of this info, so the only use for it in such a configuration (when using the built in net-if) would be tracing.

For `SnmpInfo`, see the user callback function *handle_report*.

async_get(UserId, TargetName, Oids) -> {ok, ReqId} | {error, Reason}

async_get(UserId, TargetName, ContextName, Oids) -> {ok, ReqId} | {error, Reason}

async_get(UserId, TargetName, Oids, Expire) -> {ok, ReqId} | {error, Reason}

async_get(UserId, TargetName, ContextName, Oids, Expire) -> {ok, ReqId} | {error, Reason}

async_get(UserId, TargetName, ContextName, Oids, Expire, ExtraInfo) -> {ok, ReqId} | {error, Reason}

Types:

UserId = term()

TargetName = target_name()

ContextName = string()

Oids = [oid()]

Expire = integer()

ExtraInfo = term()

ReqId = term()

Reason = term()

Asynchronous `get-request`.

The reply, if it arrives, will be delivered to the user through a call to the snmpm_user callback function `handle_pdu`.

The `Expire` time indicates for how long the request is valid (after which the manager is free to delete it).

`ExtraInfo` is an opaque data structure passed on to the net-if process. The net-if process included in this application makes no use of this info, so the only use for it in such a configuration (when using the built in net-if) would be tracing.

sync_get_next(UserId, TargetName, Oids) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_get_next(UserId, TargetName, ContextName, Oids) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_get_next(UserId, TargetName, Oids, Timeout) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_get_next(UserId, TargetName, ContextName, Oids, Timeout) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_get_next(UserId, TargetName, ContextName, Oids, Timeout, ExtraInfo) -> {ok, SnmpReply, Remaining} | {error, Reason}

Types:

UserId = term()

TargetName = target_name()

ContextName = string()

- **Oids = [oid()]**
- **Timeout = integer()**
- **ExtraInfo = term()**
- **SnmpReply = snmp_reply()**
- **Remaining = integer()**
- **Reason = {send_failed, ReqId, R} | {invalid_sec_info, SecInfo, SnmpInfo} | term()**
- **R = term()**

Synchronous `get-next-request`.

`Remaining` time of the given or default timeout time.

When *Reason* is *{send_failed, ...}* it means that the net_if process failed to send the message. This could happen because of any number of reasons, i.e. encoding error. *R* is the actual reason in this case.

`ExtraInfo` is an opaque data structure passed on to the net-if process. The net-if process included in this application makes no use of this info, so the only use for it in such a configuration (when using the built in net-if) would be tracing.

`async_get_next(UserId, TargetName, Oids) -> {ok, ReqId} | {error, Reason}`

`async_get_next(UserId, TargetName, ContextName, Oids) -> {ok, ReqId} | {error, Reason}`

`async_get_next(UserId, TargetName, Oids, Expire) -> {ok, ReqId} | {error, Reason}`

`async_get_next(UserId, TargetName, ContextName, Oids, Expire) -> {ok, ReqId} | {error, Reason}`

`async_get_next(UserId, TargetName, ContextName, Oids, Expire, ExtraInfo) -> {ok, ReqId} | {error, Reason}`

Types:

- **UserId = term()**
- **TargetName = target_name()**
- **ContextName = string()**
- **Oids = [oid()]**
- **Expire = integer()**
- **ExtraInfo = term()**
- **ReqId = integer()**
- **Reason = term()**

Asynchronous `get-next-request`.

The reply will be delivered to the user through a call to the snmpm_user callback function `handle_pdu`.

The `Expire` time indicates for how long the request is valid (after which the manager is free to delete it).

sync_set(UserId, TargetName, VarsAndVals) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_set(UserId, TargetName, ContextName, VarsAndVals) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_set(UserId, TargetName, VarsAndVals, Timeout) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_set(UserId, TargetName, ContextName, VarsAndVals, Timeout) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_set(UserId, TargetName, ContextName, VarsAndVals, Timeout, ExtraInfo) -> {ok, SnmpReply, Remaining} | {error, Reason}

Types:

UserId = term()

TargetName = target_name()

ContextName = string()

VarsAndVals = vars_and_vals()

Timeout = integer()

ExtraInfo = term()

SnmpReply = snmp_reply()

Remaining = integer()

Reason = {send_failed, ReqId, ActualReason} | {invalid_sec_info, SecInfo, SnmpInfo} | term()

ActualReason = term()

Synchronous `set-request`.

`Remaining` time of the given or default timeout time.

When *Reason* is *{send_failed, ...}* it means that the net_if process failed to send the message. This could happen because of any number of reasons, i.e. encoding error. *R* is the actual reason in this case.

When *var_and_val()* is *{oid(), value()}*, the manager makes an educated guess based on the loaded mibs.

`ExtraInfo` is an opaque data structure passed on to the net-if process. The net-if process included in this application makes no use of this info, so the only use for it in such a configuration (when using the built in net-if) would be tracing.

async_set(UserId, TargetName, VarsAndVals) -> {ok, ReqId} | {error, Reason}

async_set(UserId, TargetName, ContextName, VarsAndVals) -> {ok, ReqId} | {error, Reason}

async_set(UserId, TargetName, VarsAndVals, Expire) -> {ok, ReqId} | {error, Reason}

async_set(UserId, TargetName, ContextName, VarsAndVals, Expire) -> {ok, ReqId} | {error, Reason}

async_set(UserId, TargetName, ContextName, VarsAndVals, Expire, ExtraInfo) -> {ok, ReqId} | {error, Reason}

Types:

UserId = term()

TargetName = target_name()

VarsAndVals = vars_and_vals()

Expire = integer()

ExtraInfo = term()

ReqId = term()

Reason = term()

Asynchronous `set-request`.

The reply will be delivered to the user through a call to the snmpm_user callback function `handle_pdu`.

The `Expire` time indicates for how long the request is valid (after which the manager is free to delete it).

When *var_and_val()* is *{oid(), value()}*, the manager makes an educated guess based on the loaded mibs.

`ExtraInfo` is an opaque data structure passed on to the net-if process. The net-if process included in this application makes no use of this info, so the only use for it in such a configuration (when using the built in net-if) would be tracing.

sync_get_bulk(UserId, TragetName, NonRep, MaxRep, Oids) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_get_bulk(UserId, TragetName, NonRep, MaxRep, ContextName, Oids) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_get_bulk(UserId, TragetName, NonRep, MaxRep, Oids, Timeout) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_get_bulk(UserId, TragetName, NonRep, MaxRep, ContextName, Oids, Timeout) -> {ok, SnmpReply, Remaining} | {error, Reason}

sync_get_bulk(UserId, TragetName, NonRep, MaxRep, ContextName, Oids, Timeout, ExtraInfo) -> {ok, SnmpReply, Remaining} | {error, Reason}

Types:

UserId = term()

TargetName = target_name()

NonRep = integer()

MaxRep = integer()

ContextName = string()

Oids = [oid()]

Timeout = integer()

ExtraInfo = term()

SnmpReply = snmp_reply()

Remaining = integer()

Reason = {send_failed, ReqId, R} | {invalid_sec_info, SecInfo, SnmpInfo} | term()

Synchronous `get-bulk-request` (See RFC1905).

`Remaining` time of the given or default timeout time.

When *Reason* is *{send_failed, ...}* it means that the net_if process failed to send the message. This could happen because of any number of reasons, i.e. encoding error. *R* is the actual reason in this case.

`ExtraInfo` is an opaque data structure passed on to the net-if process. The net-if process included in this application makes no use of this info, so the only use for it in such a configuration (when using the built in net-if) would be tracing.

async_get_bulk(UserId, TargetName, NonRep, MaxRep, Oids) -> {ok, ReqId} | {error, Reason}

async_get_bulk(UserId, TargetName, NonRep, MaxRep, ContextName, Oids) -> {ok, ReqId} | {error, Reason}

async_get_bulk(UserId, TargetName, NonRep, MaxRep, Oids, Expire) -> {ok, ReqId} | {error, Reason}

async_get_bulk(UserId, TargetName, NonRep, MaxRep, ContextName, Oids, Expire) -> {ok, ReqId} | {error, Reason}

async_get_bulk(UserId, TargetName, NonRep, MaxRep, ContextName, Oids, Expire, ExtraInfo) -> {ok, ReqId} | {error, Reason}

Types:

UserId = term()

TargetName = target_name()

NonRep = integer()

MaxRep = integer()

ContextName = string()

Oids = [oid()]

Expire = integer()

ExtraInfo = term()

ReqId = integer()

Reason = term()

Asynchronous `get-bulk-request` (See RFC1905).

The reply will be delivered to the user through a call to the snmpm_user callback function `handle_pdu`.

The `Expire` time indicates for how long the request is valid (after which the manager is free to delete it).

`cancel_async_request(UserId, ReqId) -> ok | {error, Reason}`

Types:

UserId = term()

ReqId = term()

Reason = term()

Cancel a previous asynchronous request.

`log_to_txt(LogDir, Mibs)`

`log_to_txt(LogDir, Mibs, OutFile) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile, Start) -> ok | {error, Reason}`

`log_to_txt(LogDir, Mibs, OutFile, LogName, LogFile, Start, Stop) -> ok | {error, Reason}`

Types:

LogDir = string()

Mibs = [MibName]

MibName = string()

OutFile = string()

LogName = string()

LogFile = string()

Start = Stop = null | datetime() | {local_time,datetime()} | {universal_time,datetime()}

Reason = disk_log_open_error() | file_open_error() | term()

disk_log_open_error() = {LogName, term()}

file_open_error() = {OutFile, term()}

Converts an Audit Trail Log to a readable text file. `OutFile` defaults to "./snmpm_log.txt". `LogName` defaults to "snmpm_log". `LogFile` defaults to "snmpm.log". See *snmp:log_to_txt* for more info.

`change_log_size(NewSize) -> ok | {error, Reason}`

Types:

NewSize = {MaxBytes, MaxFiles}

MaxBytes = integer()

MaxFiles = integer()

Reason = term()

Changes the log size of the Audit Trail Log. The application must be configured to use the audit trail log function. Please refer to disk_log(3) in Kernel Reference Manual for a description of how to change the log size.

The change is permanent, as long as the log is not deleted. That means, the log size is remembered across reboots.

set_log_type(NewType) -> {ok, OldType} | {error, Reason}

Types:

NewType = OldType = atl_type()

Reason = term()

Changes the run-time Audit Trail log type.

Note that this has no effect on the application configuration as defined by configuration files, so a node restart will revert the config to whatever is in those files.

This function is primarily useful in testing/debugging scenarios.

load_mib(Mib) -> ok | {error, Reason}

Types:

Mib = MibName

MibName = string()

Reason = term()

Load a `Mib` into the manager. The `MibName` is the name of the Mib, including the path to where the compiled mib is found. For example,

```
        Dir = code:priv_dir(my_app) ++ "/mibs/",
        snmpm:load_mib(Dir ++ "MY-MIB").
```

unload_mib(Mib) -> ok | {error, Reason}

Types:

Mib = MibName

MibName = string()

Reason = term()

Unload a `Mib` from the manager. The `MibName` is the name of the Mib, including the path to where the compiled mib is found. For example,

```
        Dir = code:priv_dir(my_app) ++ "/mibs/",
        snmpm:unload_mib(Dir ++ "MY-MIB").
```

which_mibs() -> Mibs

Types:

Mibs = [{MibName, MibFile}]

MibName = atom()

MibFile = string()

Get a list of all the mib's loaded into the manager.

name_to_oid(Name) -> {ok, Oids} | {error, Reason}

Types:

Name = atom()

Oids = [oid()]

Transform a alias-name to its oid.

Note that an alias-name is only unique within the mib, so when loading several mib's into a manager, there might be several instances of the same aliasname.

`oid_to_name(Oid) -> {ok, Name} | {error, Reason}`

Types:

Oid = oid()

Name = atom()

Reason = term()

Transform a oid to its aliasname.

`oid_to_type(Oid) -> {ok, Type} | {error, Reason}`

Types:

Oid = oid()

Type = atom()

Reason = term()

Retreive the type (asn1 bertype) of an oid.

`backup(BackupDir) -> ok | {error, Reason}`

Types:

BackupDir = string()

Backup persistent data handled by the manager.

BackupDir cannot be identical to DbDir.

`info() -> [{Key, Value}]`

Types:

Key = atom()

Value = term()

Returns a list (a dictionary) containing information about the manager. Information includes statistics counters, miscellaneous info about each process (e.g. memory allocation), and so on.

`verbosity(Ref, Verbosity) -> void()`

Types:

Ref = server | config | net_if | note_store | all

Verbosity = verbosity()

verbosity() = silence | info | log | debug | trace

Sets verbosity for the designated process. For the lowest verbosity `silence`, nothing is printed. The higher the verbosity, the more is printed.

`format_reason(Reason) -> string()`

`format_reason(Prefix, Reason) -> string()`

Types:

Reason = term()

Prefix = integer() | string()

This utility function is used to create a formatted (pretty printable) string of the error reason received from either:

- The `Reason` returned value if any of the sync/async get/get-next/set/get-bulk functions returns `{error, Reason}`
- The `Reason` parameter in the *handle_error* user callback function.

`Prefix` should either be an indention string (e.g. a list of spaces) or a positive integer (which will be used to create the indention string of that length).

snmpm_conf

Erlang module

The module `snmpm_conf` contains various utility functions to used for manipulating (write/append/read) the config files of the SNMP manager.

Exports

`manager_entry(Tag, Val) -> manager_entry()`

Types:

Tag = address | port | engine_id | max_message_size

Val = term()

manager_entry() = term()

Create an entry for the manager config file, `manager.conf`.

The type of `Val` depends on the value of `Tag`, see *Manager Information* for more info.

`write_manager_config(Dir, Conf) -> ok`

`write_manager_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [manager_entry()]

Write the manager config to the manager config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Manager Information* for more info.

`append_manager_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [manager_entry()]

Append the config to the current manager config file.

`Dir` is the path to the directory where to store the config file.

See *Manager Information* for more info.

`read_manager_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [manager_entry()]

Read the current manager config file.

`Dir` is the path to the directory where to store the config file.

See *Manager Information* for more info.

`users_entry(UserId) -> users_entry()`

`users_entry(UserId, UserMod) -> users_entry()`

`users_entry(UserId, UserMod, UserData) -> users_entry()`

Types:

UserId = term()

UserMod = atom()

UserData = term()

standard_entry() = term()

Create an entry for the manager users config file, `users.conf`.

`users_entry(UserId)` translates to the following call: `users_entry(UserId, snmpm_user_default)`.

`users_entry(UserId, UserMod)` translates to the following call: `users_entry(UserId, UserMod, undefined)`.

See *Users* for more info.

`write_users_config(Dir, Conf) -> ok`

`write_users_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [users_entry()]

Write the manager users config to the manager users config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Users* for more info.

`append_users_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [users_entry()]

Append the users config to the current manager users config file.

`Dir` is the path to the directory where to store the config file.

See *Users* for more info.

`read_users_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [users_entry()]

Read the current manager users config file.

`Dir` is the path to the directory where to store the config file.

See *Users* for more info.

`agents_entry(UserId, TargetName, Comm, Ip, Port, EngineID, Timeout, MaxMessageSize, Version, SecModel, SecName, SecLevel) -> agents_entry()`

Types:

UserId = term()

TargetName = string()

Comm = string()

Ip = string()

Port = integer()

EngineID = string()

Timeout = integer()

MaxMessageSize = integer()

Version = v1 | v2 | v3

SecModel = v1 | v2c | usm

SecName = string()

SecLevel = noAuthNoPriv | authNoPriv | authPriv

agents_entry() = term()

Create an entry for the manager agents config file, `agents.conf`.

See *Agents* for more info.

`write_agents_config(Dir, Conf) -> ok`

`write_agents_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [_entry()]

Write the manager agents config to the manager agents config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Agents* for more info.

`append_agents_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [agents_entry()]

Append the agents config to the current manager agents config file.

`Dir` is the path to the directory where to store the config file.

See *Agents* for more info.

`read_agents_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [agents_entry()]

Read the current manager agents config file.

`Dir` is the path to the directory where to store the config file.

See *Agents* for more info.

`usm_entry(EngineID, UserName, AuthP, AuthKey, PrivP, PrivKey) -> usm_entry()`

`usm_entry(EngineID, UserName, SecName, AuthP, AuthKey, PrivP, PrivKey) -> usm_entry()`

Types:

EngineID = string()

UserName = string()

SecName = string()

AuthP = usmNoAuthProtocol | usmHMACMD5AuthProtocol | usmHMACSHAAuthProtocol

AuthKey = [integer()]

PrivP = usmNoPrivProtocol | usmDESPrivProtocol | usmAesCfb128Protocol

PrivKey = [integer()]

usm_entry() = term()

Create an entry for the agent community config file, `community.conf`.

See *Security data for USM* for more info.

`write_usm_config(Dir, Conf) -> ok`

`write_usm_config(Dir, Hdr, Conf) -> ok`

Types:

Dir = string()

Hdr = string()

Conf = [usm_entry()]

Write the manager usm config to the manager usm config file.

`Dir` is the path to the directory where to store the config file.

`Hdr` is an optional file header (note that this text is written to the file as is).

See *Security data for USM* for more info.

`append_usm_config(Dir, Conf) -> ok`

Types:

Dir = string()

Conf = [usm_entry()]

Append the usm config to the current manager usm config file.

`Dir` is the path to the directory where to store the config file.

See *Security data for USM* for more info.

`read_usm_config(Dir) -> Conf`

Types:

Dir = string()

Conf = [usm_entry()]

Read the current manager usm config file.

`Dir` is the path to the directory where to store the config file.

See *Security data for USM* for more info.

snmpm_mpd

Erlang module

The module `snmpm_mpd` implements the version independent Message Processing and Dispatch functionality in SNMP for the manager. It is supposed to be used from a Network Interface process (*Definition of Manager Net if*).

Exports

`init_mpd(Vsns) -> mpd_state()`

Types:

Vsns = [Vsn]

Vsn = v1 | v2 | v3

This function can be called from the net_if process at start-up. The options list defines which versions to use.

It also initializes some SNMP counters.

`process_msg(Msg, TDomain, Addr, Port, State, NoteStore, Logger) -> {ok, Vsn, Pdu, PduMS, MsgData} | {discarded, Reason}`

Types:

Msg = binary()

TDomain = snmpUDPDomain

Addr = {integer(), integer(), integer(), integer()}

Port = integer()

State = mpd_state()

NoteStore = pid()

Logger = function()

Vsn = 'version-1' | 'version-2' | 'version-3'

Pdu = #pdu

PduMs = integer()

MsgData = term()

Processes an incoming message. Performs authentication and decryption as necessary. The return values should be passed the manager server.

`NoteStore` is the `pid()` of the note-store process.

`Logger` is the function used for audit trail logging.

In the case when the pdu type is `report`, `MsgData` is either `ok` or `{error, ReqId, Reason}`.

`generate_msg(Vsn, NoteStore, Pdu, MsgData, Logger) -> {ok, Packet} | {discarded, Reason}`

Types:

Vsn = 'version-1' | 'version-2' | 'version-3'

NoteStore = pid()

Pdu = #pdu

MsgData = term()

Logger = function()

Packet = binary()

Reason = term()

Generates a possibly encrypted packet to be sent to the network.

`NoteStore` is the `pid()` of the note-store process.

`MsgData` is the message specific data used in the SNMP message. In SNMPv1 and SNMPv2c, this message data is the community string. In SNMPv3, it is the context information.

`Logger` is the function used for audit trail logging.

generate_response_msg(Vsn, Pdu, MsgData, Logger) -> {ok, Packet} | {discarded, Reason}

Types:

Vsn = 'version-1' | 'version-2' | 'version-3'

Pdu = #pdu

MsgData = term()

Logger = function()

Packet = binary()

Reason = term()

Generates a possibly encrypted response packet to be sent to the network.

`MsgData` is the message specific data used in the SNMP message. This value is received from the *process_msg* function.

snmpm_network_interface

Erlang module

This module defines the behaviour of the manager network interface. A `snmpm_network_interface` compliant module must export the following functions:

- *start_link/2*
- *stop/1*
- *send_pdu/7*
- *inform_response/4*
- *note_store/2*
- *info/1*
- *get_log_type/1*
- *set_log_type/2*
- *verbosity/2*

The semantics of them and their exact signatures are explained below.

Exports

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

Types:

Server = pid()

NoteStore = pid()

Start-link the network interface process.

`Server` is the pid of the managing process.

`NoteStore` is the pid of the note-store process.

`stop(Pid) -> void()`

Types:

Pid = pid()

Stop the network interface process.

`send_pdu(Pid, Pdu, Vsn, MsgData, Addr, Port, ExtraInfo) -> void()`

Types:

Pid = pid()

Pdu = pdu()

Vsn = 'version-1' | 'version-2' | 'version-3'

MsgData = term()

Addr = address()

Port = integer()

ExtraInfo = term()

Request the network interface process (`Pid`) to send this pdu (`Pdu`).

`ExtraInfo` is some opaque data that is passed to the net-if process. It originates from the `ExtraInfo` parameter in the calls to the *synchronous get-request*, *asynchronous get-request*, *synchronous get-next-request*, *asynchronous get-next-request*, *synchronous set-request* and *asynchronous set-request* functions. Whether the net-if process chooses to use this is implementation dependent. The net-if process included in this application ignores it.

`inform_response(Pid, Ref, Addr, Port) -> void()`

Types:

Pid = pid()

Ref = term()

Addr = address()

Port = integer()

Instruct the network interface process to send the response (acknowledgment) to an inform-request.

`Ref` is something that can be used to identify the inform-request, e.g. request-id of the inform-request.

`Addr` and `Port` identifies the agent, from which the inform-request originated.

`note_store(Pid, NoteStore) -> void()`

Types:

Pid = pid()

NoteStore = pid()

Change the pid of the note-store process. This is used when the server re-starts the note_store (e.g. after a crach).

`info(Pid) -> [{Key, Value}]`

Types:

Pid = pid()

The info returned is basically up to the implementer to decide. The implementation provided by this application provides info about memory allocation and various socket information.

The info returned by this function is returned together with other info collected by the manager when the *info* function is called (tagged with the key `net_if`).

`verbosity(Pid, Verbosity) -> void()`

Types:

Pid = pid()

Verbosity = verbosity()

Change the verbosity of the network interface process.

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

Types:

Pid = pid()

LogType = atl_type()

Reason = term()

The Audit Trail Log is managed by the network interface process. So, it is this process that has to return the actual log-type.

`set_log_type(Pid, NewType) -> {ok, OldType} | {error, Reason}`

Types:

Pid = pid()

NewType = OldType = atl_type()

Reason = term()

The Audit Trail Log is managed by the network interface process. So, it is this process that has to do the actual changing of the type.

See *set_log_type* for more info.

snmpm_user

Erlang module

This module defines the behaviour of the manager user. A `snmpm_user` compliant module must export the following functions:

- handle_error/3
- handle_agent/4
- handle_pdu/4
- handle_trap/3
- handle_inform/3
- handle_report/3

The semantics of them and their exact signatures are explained below.

Note that if an agent is registered using the old, no longer documented, functions (using Addr and Port), the old variant of the callback functions, handle_pdu, handle_trap, handle_inform and handle_report, will be called.

Exports

`handle_error(ReqId, Reason, UserData) -> Reply`

Types:

ReqId = integer()

Reason = {unexpected_pdu, SnmpInfo} | {invalid_sec_info, SecInfo, SnmpInfo} | {empty_message, Addr, Port} | term()

Addr = ip_address()

Port = integer()

UserData = term()

Reply = ignore

This function is called when the manager needs to communicate an "asynchronous" error, to the user: e.g. failure to send an asynchronous message (i.e. encoding error), a received message was discarded due to security error, the manager failed to generate a response message to a received inform-request, or when receiving an unexpected PDU from an agent (could be an expired async request).

If `ReqId` is less then 0, it means that this information was not available to the manager (that info was never retrieved before the message was discarded).

For `SnmpInfo` see handle_agent below.

`handle_agent(Addr, Port, Type, SnmpInfo, UserData) -> Reply`

Types:

Addr = ip_address()

Port = integer()

Type = pdu | trap | report | inform

SnmpInfo = SnmpPduInfo | SnmpTrapInfo | SnmpReportInfo | SnmpInformInfo

ErrorStatus = atom()

ErrorIndex = integer()

Varbinds = [varbind()]

varbind() = #varbind

UserData = term()

Reply = ignore | {register, UserId, TargetName, agent_info()}

UserId = term()

TargetName = target_name()

agent_info() = [{agent_info_item(), agent_info_value()}]

This function is called when a message is received from an unknown agent.

Note that this will always be the default user that is called.

For more info about the `agent_info()`, see *register_agent*.

The arguments `Type` and `SnmpInfo` relates in the following way:

- `pdu` - `SnmpPduInfo` (see *handle_pdu* for more info).
- `trap` - `SnmpTrapInfo` (see *handle_trap* for more info).
- `report` - `SnmpReportInfo` (see *handle_report* for more info).
- `inform` - `SnmpInformInfo` (see *handle_inform* for more info).

The only user which would return `{register, UserId, TargetName, agent_info()}` is the *default user*.

`handle_pdu(TargetName, ReqId, SnmpPduInfo, UserData) -> Reply`

Types:

TargetName = target_name()

ReqId = term()

SnmpPduInfo = {ErrorStatus, ErrorIndex, Varbinds}

ErrorStatus = atom()

ErrorIndex = integer()

Varbinds = [varbind()]

varbind() = #varbind

UserData = term()

Reply = ignore

Handle the reply to an asynchronous request, such as *async_get*, *async_get_next* or *async_set*.

It could also be a late reply to a synchronous request.

`ReqId` is returned by the asynchronous request function.

`handle_trap(TargetName, SnmpTrapInfo, UserData) -> Reply`

Types:

TargetName = TargetName2 = target_name()

SnmpTrapInfo = {Enteprise, Generic, Spec, Timestamp, Varbinds} | {ErrorStatus, ErrorIndex, Varbinds}

Enterprise = oid()

Generic = integer()

Spec = integer()

Timestamp = integer()

ErrorStatus = atom()

ErrorIndex = integer()

Varbinds = [varbind()]

varbind() = #varbind

UserData = term()

Reply = ignore | unregister | {register, UserId, TargetName2, agent_info()}

UserId = term()

agent_info() = [{agent_info_item(), agent_info_value()}]

Handle a trap/notification message from an agent.

For more info about the `agent_info()`, see *register_agent*

The only user which would return `{register, UserId, TargetName2, agent_info()}` is the *default user*.

`handle_inform(TargetName, SnmpInformInfo, UserData) -> Reply`

Types:

TargetName = TargetName2 = target_name()

SnmpInformInfo = {ErrorStatus, ErrorIndex, Varbinds}

ErrorStatus = atom()

ErrorIndex = integer()

Varbinds = [varbind()]

varbind() = #varbind

UserData = term()

Reply = ignore | unregister | {register, UserId, TargetName2, agent_info()}

UserId = term()

agent_info() = [{agent_info_item(), agent_info_value()}]

Handle a inform message.

For more info about the `agent_info()`, see *register_agent*

The only user which would return `{register, UserId, TargetName2, agent_info()}` is the *default user*.

If the *inform request behaviour* configuration option is set to `user` or `{user, integer()}`, the response (acknowledgment) to this inform-request will be sent when this function returns.

`handle_report(TargetName, SnmpReportInfo, UserData) -> Reply`

Types:

TargetName = TargetName2 = target_name()

Addr = ip_address()

Port = integer()

SnmpReportInfo = {ErrorStatus, ErrorIndex, Varbinds}

ErrorStatus = atom()

ErrorIndex = integer()

Varbinds = [varbind()]

varbind() = #varbind

UserData = term()

Reply = ignore | unregister | {register, UserId, TargetName2, agent_info()}

UserId = term()

agent_info() = [{agent_info_item(), agent_info_value()}]

Handle a report message.

For more info about the `agent_info()`, see *register_agent*

The only user which would return `{register, UserId, TargetName2, agent_info()}` is the *default user*.

snmpm_network_interface_filter

Erlang module

This module defines the behaviour of the manager network interface filter. A `snmpm_network_interface_filter` compliant module must export the following functions:

- *accept_recv/2*
- *accept_send/2*
- *accept_recv_pdu/3*
- *accept_send_pdu/2*

The semantics of them and their exact signatures are explained below.

The purpose of the network interface filter is to allow for filtering of messages (accept or reject) receive and send. This is done on two levels:

- The first level is at the UDP entry / exit point, i.e. immediately after the receipt of the message, before any message processing is done (accept_recv) and immediately before sending the message, after all message processing is done (accept_send).
- The second level is at the MPD entry / exit point, i.e. immediately after the basic message processing (accept_recv_pdu) / immediately before the basic message processing (accept_send_pdu).

Note that the network interface filter is something which is used by the network interface implementation provided by the application (`snmpm_net_if`). The default filter accepts all messages.

A network interface filter can e.g. be used during testing or for load regulation.

DATA TYPES

```
port() = integer() > 0
pdu_type() = 'get-request' | 'get-next-request' | 'get-response' | 'set-request' | trap | 'get-bulk-request' |
```

Exports

`accept_recv(Addr, Port) -> boolean()`

Types:

Addr = ip_address()

Port = port()

Called at the reception of a message (before *any* processing has been done).

For the message to be rejected, the function *must* return *false*.

`accept_send(Addr, Port) -> boolean()`

Types:

Addr = ip_address()

Port = port()

Called before the sending of a message (after *all* processing has been done).

For the message to be rejected, the function *must* return *false*.

accept_recv_pdu(Addr, Port, PduType) -> boolean()

Types:

Addr = ip_address()

Port = port()

PduType = pdu_type()

Called after the basic message processing (MPD) has been done, but before the pdu is handed over to the server for primary processing.

For the pdu to be rejected, the function *must* return *false*.

accept_send_pdu(Addr, Port, PduType) -> boolean()

Types:

Addr = ip_address()

Port = port()

PduType = pdu_type() > 0

Called before the basic message processing (MPD) is done, when a pdu has been received from the master-agent.

For the message to be rejected, the function *must* return *false*.