Simple Network Management Protocol (SNMP)

version 3.4

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

Chapter 1

SNMP User's Guide

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

1.1 SNMP Introduction

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

- 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 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 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 Subagents and MIB loading, Internal MIBs, and Traps.
- Chapter 3: “Instrumentation Functions” describes how instrumentation functions should be defined in Erlang for the different operations.
- Chapter 4: “The MIB Compiler” describes the features and the operation of the MIB compiler.
- Chapter 5: “Running the Agent” describes how to start and configure the agent. Topics on how to debug the agent are also included.
- Chapter 6: “Implementation Example” describes how an MIB can be implemented with the SNMP Development Toolkit. Implementation examples are included.
- Chapter 7: “Advanced Topics” describes subagents, agent semantics, audit trail logging, and the consideration of distributed tables.
- Chapter 8: “Definition of Configuration Files” is a reference chapter, which contains more detailed information about the configuration files.
- Chapter 9: “Definition of Instrumentation Functions” is a reference chapter which contains more detailed information about the instrumentation functions.
- Chapter 10: “Definition of Net if” is a reference chapter, which describes the Net if function in detail.
- Appendix A describes the conversion of SNMPv2 to SNMPv1 error messages.
- Appendix B contains the RFC1903 text on `RowStatus`.

1.1.4 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 Functional Description

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 MIB compiler, which understands SMIv1 (RFC1155, 1212, and 1215) and SMIv2 (RFC1902, 1903, and 1904).
- A multi-lingual SNMP manager, which can be used for simple interactive testing and for writing test suites.

The SNMP agent system consists of one Master Agent and optional Subagents.

The tool makes it easy to dynamically extend an SNMP agent in runtime. MIBs can be loaded and unloaded at any time. It is also easy to change the implementation of an MIB in runtime, 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 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 systrem.

1.2.2 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 runtime
- 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 subagent concept via distributed Erlang
- a mechanism for sending notifications (traps and informs)
- support for implementing SNMP tables in the Mnesia DBMS.

1.2.3 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.

1.2.4 Operation

The following steps are needed to get a running agent:

1. Write your MIB in SMI in a text file.
2. Write the instrumentation functions in Erlang and compile them.
3. Put their names in the association file.
4. Run the MIB together with the association file through the MIB compiler.

5. Configure the agent.
6. Start the agent.
7. Load the compiled MIB into the agent.

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

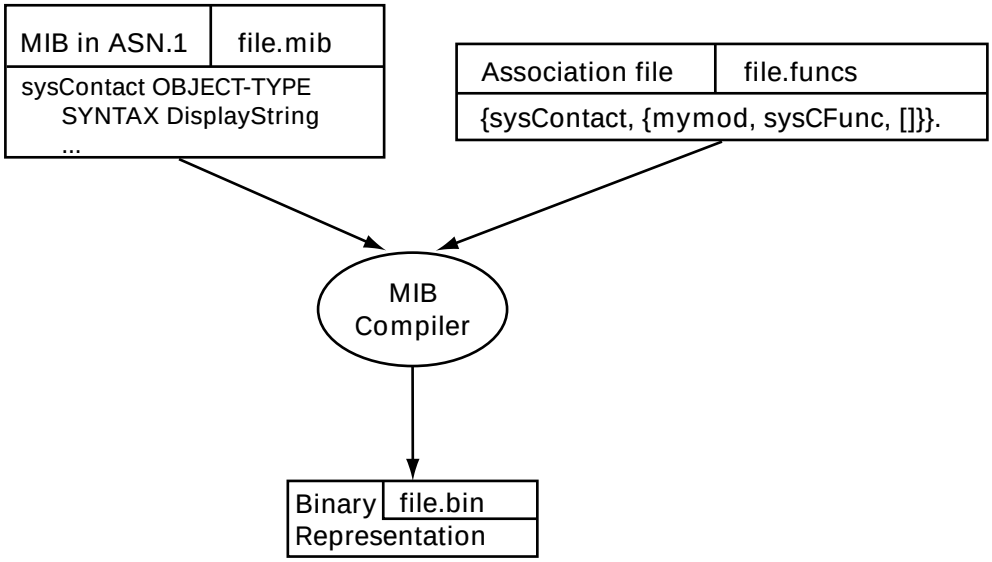


Figure 1.1: MIB Compiler Principles

The compiler parses the SMI file and associates each table or variable with an instrumentation function (see the figure MIB Compiler Principles [page 6]). 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 [page 6]). The instrumentation is ordinary Erlang code which is loaded explicitly or automatically the first time it is called.

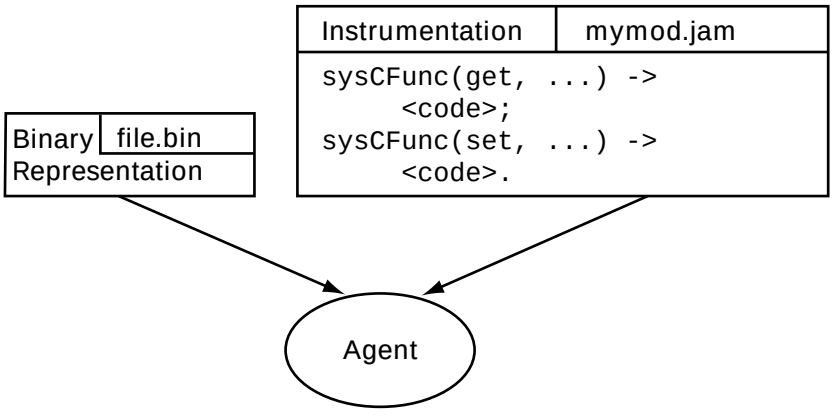


Figure 1.2: Starting the Agent

The SNMP agent system consists of one Master Agent and optional subagents. The Master Agent can be seen as a special kind of subagent. 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 subagent.

Subagents are only needed if your application requires special support for distribution from the SNMP toolkit. A subagent 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.

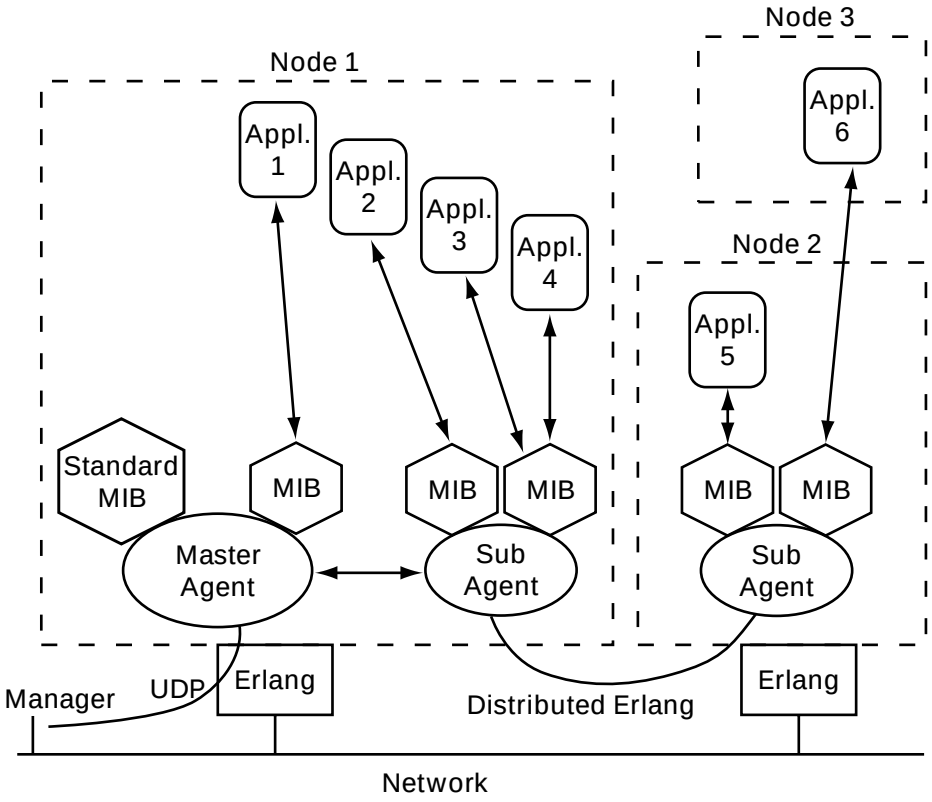


Figure 1.3: Architecture

A typical operation could include the following steps:

1. The Manager sends a request to the Agent.
2. The Master Agent decodes the incoming UDP packet.
3. The Master Agent determines which items in the request that should be processed here and which items should be forwarded to its subagent.
4. Step 3 is repeated by all subagents.
5. Each subagent calls the instrumentation for its loaded MIBs.
6. The results of calling the instrumentation are propagated back to the Master Agent.
7. 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.
- Subagents can also have subagents. Each subagent can have an arbitrary number of child subagents 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.

1.2.5 Subagents 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 subagent, which is controlled by the application and is able to register and de-register itself. A subagent can register itself for maniging a sub-tree (not to be mixed up with `erlang:register`). The sub-tree is identified by an Object Identifier. When a subagent 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 subagent can be started and stopped at any time.

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

Subagents 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 runtime
- to provide interaction with other SNMP agent toolkits.

Refer to the chapter Advanced Topics [page 39] in this User's Guide for more information about these topics.

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

1.2.6 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 `snmp: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 Configuration Files [page 27].

1.2.7 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 distributionand are placed in the directories `mibs`, `src`, and `include`, respectively, in the `snmp` application.

The `.hrl` files are generated with `snmp: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 startup from a set of configuration files. These are described in Configuration Files [page 27].

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 `snmp_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 [page 27].)

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 startup.

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.

The SNMP-NOTIFICATION-MIB is implemented according to `snmpNotifyBasicCompliance`. It means, the notification filtering is not implemented.

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:

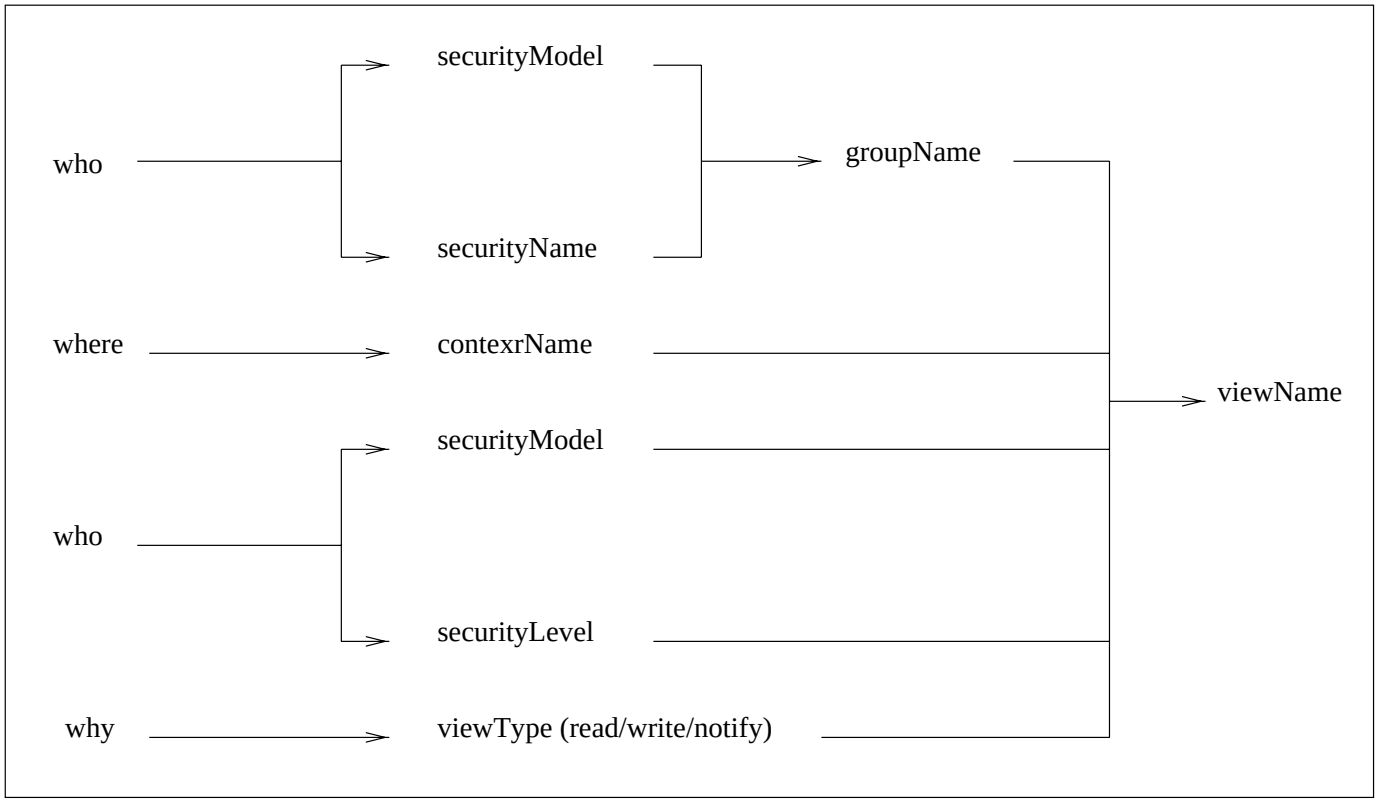


Figure 1.4: Overview of the mechanism of MIB selection

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.

1.2.8 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:

```
snmp:send_notification(Agent,Notification,Receiver
                       [,NotifyName,ContextName,Varbinds])
snmp: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 `snmp: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 acknowledgements 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 `snmp: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.

Subagent Path

If a value for an object is not given to the send_notification function, the subagent will perform a get-operation to retrieve it. If the object is not implemented in this subagent, 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.

1.3 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 calles to get-next.

1.3.1 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 [page 50].

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 1.5: 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:

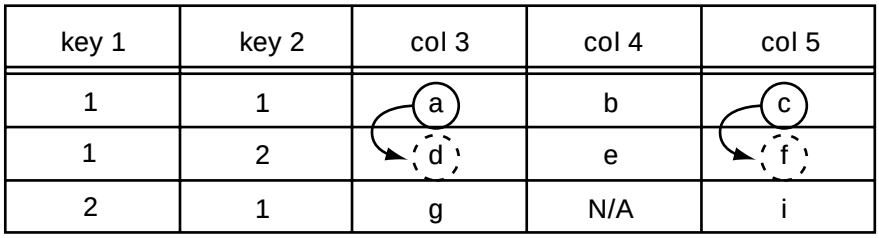


Figure 1.6: GetNext from [3,1,1] and [5,1,1].

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:

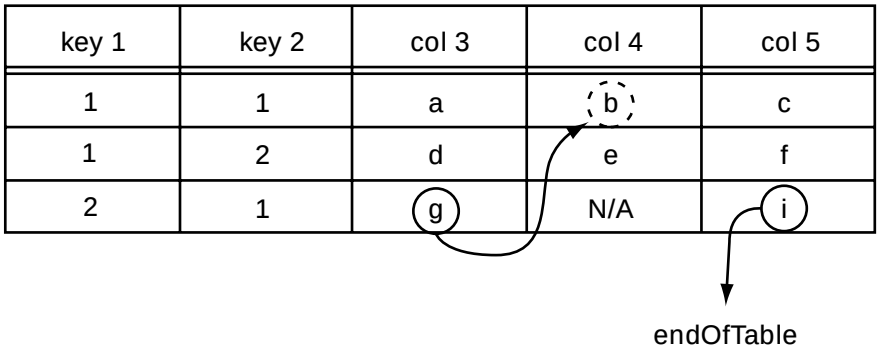


Figure 1.7: GetNext from [3,2,1] and [5,2,1].

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:

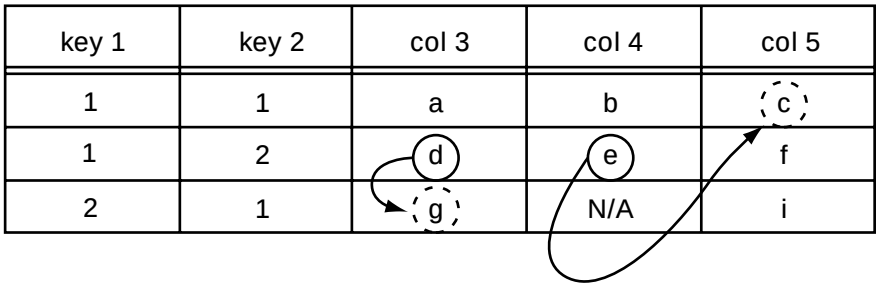


Figure 1.8: GetNext from [3,1,2] and [4,1,2].

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.

1.3.2 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]}}.
```

1.3.3 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.

1.3.4 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.

1.4 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 [page 22] for more details).

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

```
snmp:c("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 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:

```
snmp:c("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:

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

finds the lastest 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>snmp: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>snmp: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 snmp c <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,true`}.

1.5 Running the Agent

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

- configuration directories and parameters
- modifying the configuration files
- starting the agent
- debugging the agent.

Refer also to the chapter Definition of Configuration Files [page 46] which contains more detailed information about the configuration files.

1.5.1 Configuring the Agent

The following two directories must exist in the system:

- the *configuration directory* stores all configuration files (refer to the chapter Definition of Configuration Files [page 46] for more information).
- the *database directory* stores the internal database files.

The agent 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:

`audit_trail_log = false | write_log | read_write_log` <optional> Specifies if an audit trail log should be used. The `disk_log` module is used to maintain a wrap log. If `write_log` is specified, only set requests are logged. If `read_write_log`, all requests are logged. Default is `false`.

`audit_trail_log_dir = string()` <optional> Specifies where the audit trail log should be stored. If `audit_trail_log` specifies that logging should take place, this parameter must be defined.

`audit_trail_log_size = {MaxBytes, MaxFiles}` <optional> 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.

`bind_to_ip_address = bool()` <optional> If `true` the agent binds to the agent IP adress. If `false` the agent listens on any IP address on the host where it is running. Default is `false`.

`force_config_load = bool()` <optional> If `true` the configuration files are re-read during startup, 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`.

`no_reuse_address = bool()` <optional> If `true` the agent does not specify that the IP and port address should be reusable. If `false` the agent the address is set to reusable. Default is `false`.

`snmp_agent_type = master | sub` <optional> If `master`, one master agent is started. Otherwise, no agents are started. Default is `master`.

`snmp_config_dir = string()` <mandatory> Defines where the SNMP configuration files and the compiled master agent MIB files are stored.

`snmp_db_dir = string()` <mandatory> Defines where the SNMP internal db files are stored.

`snmp_master_agent_mibs = [string()]` <optional> Specifies a list of MIB names and defines which MIBs are initially loaded into the SNMP master agent. These MIBs are loaded from `snmp_config_dir`.

snmp_multi_threaded = bool() <optional> If true, the agent is multi-threaded, with one thread for each get request. Default is false.

snmp_req_limit = integer() | infinity <optional> The number of simultaneous requests (get, get-next and get-bulk) the agent will allow. Default is infinity.

snmp_priority = atom() <optional> Defines the Erlang priority for all SNMP processes. Default is normal.

v1 = bool() <optional> Defines if the agent shall speak SNMPv1. Default is true.

v2 = bool() <optional> Defines if the agent shall speak SNMPv2c. Default is true.

v3 = bool() <optional> Defines if the agent shall speak SNMPv3. Default is true.

snmp_local_db_auto_repair = false | true | true_verbose <optional> When starting snmp_local_db it always tries to open an existing database. If false, and some errors occur, a new datebase is created instead. If true, erroneous transactions (in the logfile) are ignored. If true_verbose, erroneous transactions (in the logfile) are igored and an error message is written. Default is true.

snmp_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 perpose of the check is to prevent that the same symbolic mibentry name is used for in different oid's. Default is false.

snmp_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 perpose of the check is to prevent that the same symbolic trap name is used for in different trap's. Default is false.

snmp_error_report_mod = atom() <optional> Defines an error report module, other then the default. Two modules are provided with the toolkit: snmp_error and snmp_error_io. Default is snmp_error.

snmp_master_agent_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP master agent. Default is silence.

snmp_symbolic_store_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP symbolic store. Default is silence.

snmp_note_store_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP note store. Default is silence.

snmp_net_if_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP net if. Default is silence.

snmp_mibserver_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP mib server. Default is silence.

snmp_mib_storage = ets | {dets,Dir} | {dets,Dir,Action} | {mnesia,Nodes} | {mnesia,Nodes,Action} <optior
Specifies how info retrieved from the mibs will be stored. Default is ets.
Dir = string(). Dir is the directory where the (dets) files will be created.
Nodes = [node()]. 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 table already exist.

1.5.2 Modifying the Configuration Files

To to start the agent, the agent 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> application:start(crypto).
```

```
ok
2> snmp:config().

Simple SNMP configuration tool (v3.0)
---------------------------------------------
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

1. System name (sysName standard variable) [mbj's agent]
2. Engine ID (snmpEngineID standard variable)[mbj's engine]
3. The UDP port the agent listens to. (standard 161) [4000]
4. IP address for the agent (only used as id
   when sending traps) [dront.ericsson.se]
5. IP address for the manager (only this manager will have access
   to the agent, traps are sent to this one) [dront.ericsson.se]
6. To what UDP port at the manager should traps
   be sent (standard 162)? [5000]
7. What SNMP version should be used (1,2,3,1&2,1&2&3,2&3)? [3]
7b. Should notifications be sent as traps or informs? [trap]
8. Do you want a none- minimum- or semi-secure configuration?
   Note that if you chose v1 or v2, you will not get any security for these
   requests (none, minimum, semi) [minimum]
8b. Give a password of at least length 8. It is used to generate
private keys for the  configuration.secretpasswd
9. Where is the configuration directory (absolute)? [/home/mbj/snmp_conf]
10. Current configuration files will now be overwritten. Ok [y]/n?
------------------------
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.
The following files were written: agent.conf, community.conf,
   standard.conf, target_addr.conf, target_params.conf,
   notify.conf vacm.conf, sys.config, usm.conf
------------------------
ok
```

1.5.3 Starting the Agent

Start Erlang with the command:

```
erl -config /home/mbj/snmp_conf/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 Agent

It is possible to debug every process of the agent (possibly with the exception of the net_if module, which could be supplied by a user of the application). This can be done in two ways. Either by calling the `snmp:verbosity/2` function or using configuration parameters [page 27]. 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`.

The *old* debugging is still available and produces more or less the same output, i.e. the debug flag can be turned on to verify that the configuration is correct and that the instrumentation functions behave as expected. The agent then shows all network communication (incoming/outgoing traffic), and calls to the instrumentation functions.

```
3> snmp:debug(snmp_master_agent, true).
ok
4>
%% Example of output from the agent when a get-next-request arrives:
** SNMP NET-IF LOG:
   got paket 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
```

Another useful function for debugging is `snmp_local_db:print/0,1,2`. For example, this function can show the counters `snmpInPkts` and `snmpOutPkts`. Enter the following command:

```
4> snmp_local_db:print().
%% A lot of information.
```

1.6 Implementation Example

The section *Implementation Example* 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.

1.6.1 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
```

1.6.2 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> snmp:c("EX1-MIB").
No accessfunction for 'friendsTable', using default.
No accessfunction for 'myName', using default.
{ok,"EX1-MIB.bin"}
3> snmp: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_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_mgr:gn([[1,3,6,1,3,7]]).
ok
* Got PDU:
[myName,0] = []
%% A set-request (now using symbolic names for convenience)
3> snmp_mgr:s([{[myName,0], "Martin"}]).
ok
* Got PDU:
[myName,0] = "Martin"
%% Try the same get-next request again
4> snmp_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_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_mgr:gn([[myName,0]]).
ok
* Got PDU:
[fName,0] = "Martin"
7> snmp_mgr:gn().
ok
* Got PDU:
[fAddress,0] = "home"
8> snmp_mgr:gn().
ok
* Got PDU:
[fStatus,0] = 1
9>
```

1.6.3 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('mbj@erlang.ericsson.se').
%% 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> snmp:c("EX1-MIB").
{ok,"EX1-MIB.bin"}
3> snmp: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 `snmp: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_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_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> snmp: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>
```

1.7 Advanced Topics

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

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

1.7.1 When to use a Subagent

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

Special Set Transaction Mechanism

Each subagent 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 subagent should be used.

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

Process Communication

A simple distributed application can be managed without subagents. The instrumentation functions can use distributed Erlang to communicate with other parts of the application. However, a subagent can be used on each node if this generates too much unnecessary traffic. A subagent 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 subagent. This subagent 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 subagent, 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 subagent. This subagent only registers and de-registers 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 subagents constructed in another package, a special subagent should be used, which communicates through a protocol specified by the other package.

1.7.2 Agent Semantics

The agent can be configured to be multi-threaded, or to process one incoming request at a time. 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 parallelize write requests 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 subagents 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 `snmp:current_request_id/0`.

1.7.3 Subagents and Dependencies

The toolkit supports the use of different types of subagents, but not the construction of subagents.

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

1.7.4 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 toolkits dedicate an SNMP subagent for each part of the table and load the corresponding MIB into all subagents. 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 subagent for each table holder. Normally, the subagent 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.

1.7.5 Fault Tolerance

The SNMP 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 subagents 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 application 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.

1.7.6 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> snmp:c("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"`.

1.7.7 Audit Trail Logging

The agent can be configured to log incoming requests and outgoing responses and traps. 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 `write_log` or a `read_write_log`. In a `write_log`, all `set` requests and their responses are stored. No `get` requests or traps are stored in a `write_log`. In a `read_write_log`, 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 `snmp:log_to_txt/2,3` for this purpose.

1.7.8 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.

1.8 Definition of 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 snmp_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 [page 9] for a description of the MIBs).

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.

1.8.1 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.

1.8.2 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.

1.8.3 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.

1.8.4 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.

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.

1.8.5 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 wildcards which match any sub-identifier. If the mask is shorter than the subtree, the tail is regarded as all ones. `null` is shorthand for a mask with all ones.

1.8.6 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 either `usmNoAuthProtocol`, `usmHMACMD5AuthProtocol`, or `usmHMACSHAAuthProtocol`.
- `AuthKeyC` is a string.
- `OwnAuthKeyC` is a string.
- `PrivP` is a `usmNoPrivProtocol` or `usmDESPrivProtocol`.

- 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 is used.

1.8.7 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.

1.8.8 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}. 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.
- TMask is a string of size 0, or size 6.
- MaxMessageSize is an integer.

1.8.9 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`.

1.9 Definition of Instrumentation Functions

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

1.9.1 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 [page 58] 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 occured. 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)

1.9.2 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 [page 58] 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 occured. 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 fumction `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 unaccessible 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 fumction `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 fumction `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 fumction `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 functionn 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.

1.10 Definition of Net if

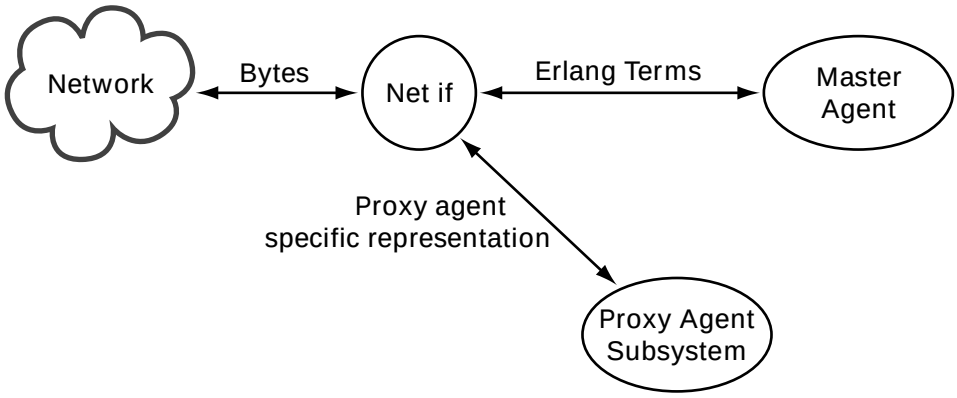


Figure 1.9: The Purpose of Net if

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 `snmp_net_if` and it uses UDP as the transport protocol.

This section describes how to write a Net if process.

1.10.1 Mandatory Functions

A Net if process must be implemented in a module that exports the `Module:start_link/2` function, which starts a new Net if process. The name of the Net if module is passed as a start argument to the `snmp_agent` process.

Function

```
Module:start_link(MasterAgent,Args)
```

Arguments

`MasterAgent` -> is a Pid.

`Args` is a a list of arguments:

- `{net_if_verbosity,silence|info|log|debug|trace}`
  A description of verbosity can be found here [page 109] and here [page 30]
- `{net_if_recbuf,integer()}`
  The size to be used for the UDP receive buffer.

Return values

The return values are:

- {`ok, Pid`}, where Pid is a linked Pid of the Net if process.
- {`error, Reason`} if the operation fails.

1.10.2 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 `snmp_mpd:process_packet` (see Reference Manual).
- `ACMData` is data used by the Access Control Module in use. Normally this is returned from `snmp_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) [page 58].
- 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 snmp_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 snmp_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 snmp_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 snmp_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 snmp_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 snmp_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 snmp_mpd for this purpose (refer to the Reference Manual, section snmp, module snmp_mpd for more details.)

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

1.11 SNMP Appendix A

1.11.1 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

continued ...

... continued

wrongLength	badValue
wrongEncoding	badValue
wrongValue	badValue
noCreation	noSuchName
inconsistentValue	badValue
resourceUnavailable	genErr
commitFailed	genErr
undoFailed	genErr
notWritable	noSuchName
inconsistentName	noSuchName

Table 1.1: Error Messages

1.12 SNMP Appendix B

1.12.1 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	
ACTION	status column does not exist	is notReady	is notInService	status column is active
set status column to createAndGo	noError or inconsistent-Value ->D	inconsist-entValue	inconsistent-Value	inconsistent-Value
set status column to createAndWait	noError or wrongValue see 1	inconsist-entValue	inconsistent-Value	inconsistent-Value
set status column to active	inconsistent-Value	inconsist-entValue	noError	noError
		or		
		see 2 ->D	->D	->D
set status column to notInService	inconsistent-Value	inconsist-entValue	noError	noError ->C
		or		or
		see 3 ->C	->C	wrongValue
set status column to destroy	noError ->A	noError ->A	noError ->A	noError ->A
set any other column to some value	see 4	noError see 1	noError ->C	see 5 ->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 absense 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)
    }
```

1.13 SNMP Release Notes

1.13.1 SNMP Development Toolkit v3.4.12

Version 3.4.12 supports code replacement in runtime from/to version 3.4.11, 3.4.10, 3.4.9, 3.4.8, 3.4.7, 3.4.6, 3.4.5, 3.4.4, 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent The SNMP agent internal data base (local db) uses dets, and does not properly handle error's from e.g. lookup.
Own Id: OTP-6210
Aux Id: Seq 10404
Aux Id: OTP-5838

Incompatibilities

-

1.13.2 SNMP Development Toolkit v3.4.11

Version 3.4.11 supports code replacement in runtime from/to version 3.4.10, 3.4.9, 3.4.8, 3.4.7, 3.4.6, 3.4.5, 3.4.4, 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent Error replies was composed with invalid OIDs for the following error counter: *usmStatsUnsupportedSecLevels* (point 5).
Own Id: OTP-5486
Aux Id: Seq 9791
Aux Id: OTP-5464

Incompatibilities

-

1.13.3 SNMP Development Toolkit v3.4.10

Version 3.4.10 supports code replacement in runtime from/to version 3.4.9, 3.4.8, 3.4.7, 3.4.6, 3.4.5, 3.4.4, 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent Error replies was composed with invalid OIDs for the following error counters: *usmStatsWrongDigests* (RFC 2574, chap 3.2, point 6), *usmStatsUnsupportedSecLevels* (point 5) and *usmStatsDecryptionErrors* (point 8a).
Own Id: OTP-5464
Aux Id: Seq 9791

agent Malformed Oid returned from a get_next operation as part of a get-bulk-request causes the agent to crash.
Own Id: OTP-5465
Aux Id: Seq 9783, Seq 9793

Incompatibilities

-

1.13.4 SNMP Development Toolkit v3.4.9

Version 3.4.9 supports code replacement in runtime from/to version 3.4.8, 3.4.7, 3.4.6, 3.4.5, 3.4.4, 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent Added missing type check for access snmpCommunityTable with is_set_ok.
Own Id: OTP-4978
Aux Id: Seq 8380

compiler Added “default value” for INTEGER with enumeration without a DEFVAL clause. The lowest valid integer value is choosen for the variable_info defval.
Own Id: OTP-5124
Aux Id: Seq 8738

Incompatibilities

-

1.13.5 SNMP Development Toolkit v3.4.8

Version 3.4.8 supports code replacement in runtime from/to version 3.4.7, 3.4.6, 3.4.5, 3.4.4, 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent Default instrumentation functions mis-behave on some, not supported, tables. Could enter infinit loop.
Own Id: OTP-5084
Aux Id: Seq 8807

Incompatibilities

-

1.13.6 SNMP Development Toolkit v3.4.7

Version 3.4.7 supports code replacement in runtime from/to version 3.4.6, 3.4.5, 3.4.4, 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

- Application test directory included in the source release.
  Own Id: OTP-5056
  Aux Id: Seq 8738

Reported Fixed Bugs and Malfunctions

compiler The value range part of the SYNTAX Integer32 does not handle values given as hexStr or bitStr (only 10-base integers).
Own Id: OTP-5051
Aux Id: Seq 8738

compiler The mib compiler cannot handle mib traps/notifications with included values (OBJECTS) which are defined later in the MIB.
Own Id: OTP-5052
Aux Id: Seq 8738

Incompatibilities

-

1.13.7 SNMP Development Toolkit v3.4.6

Version 3.4.6 supports code replacement in runtime from/to version 3.4.5, 3.4.4, 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent Incorrect (v3-) password causes the agent to not send a report back to the manager and counters to not be updated, due to a decode crash.
Martin Bjrklund
Own Id: OTP-5041

Incompatibilities

-

1.13.8 SNMP Development Toolkit v3.4.5

Version 3.4.5 supports code replacement in runtime from/to version 3.4.4, 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent snmp_user_based_sm_mib:add_user/13 calls undef function snmp_conf:check_user/1.
Should have been snmp_conf:check_usm/1. Introduced in snmp-3.4.3
Own Id: OTP-5017

Incompatibilities

-

1.13.9 SNMP Development Toolkit v3.4.4

Version 3.4.4 supports code replacement in runtime from/to version 3.4.3, 3.4.2, 3.4.1 and 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent SNMP trap send stops after change to Access Group Data.
Own Id: OTP-4999
Aux Id: Seq 8626

agent get-next on vacmAccessTable exits (badarg) for column 3.
Own Id: OTP-5000
Aux Id: Seq 8626

Incompatibilities

-

1.13.10 SNMP Development Toolkit v3.4.3

Version 3.4.3 supports code replacement in runtime from/to version 3.4.2, 3.4.1 and 3.4.

Improvements and new features

agent Improved load control. Added a new config parameter, snmp_req_limit, which allow for some load control, see configuration parameters [page 27].
Own Id: OTP-4980
Aux Id: Seq 8446

agent The example manager cannot handle start option receive_type.
Nicolas Niclausse
Own Id: OTP-4993

compiler Notifications now included in generated header files.
Own Id: OTP-4931
Aux Id: Seq 8421

compiler Defines of the SNMPv2-TC now builtin. This also means that the SNMPv2-TC provided with this application is the proper one.
Own Id: OTP-4934
Aux Id: Seq 8419

agent Added functions to add/delete config in runtime, equivalent to the config files:

- add_community/5 [page 110] and delete_community/1 [page 111]
- add_context/1 [page 115] and delete_context/1 [page 115]

- add_notify/3 [page 135] and delete_notify/1 [page 136]
- add_addr/10 [page 145], delete_addr/1 [page 145], add_params/5 [page 145] and delete_params/1 [page 146]
- add_user/13 [page 147] and delete_user/1 [page 148]
- add_sec2group/3 [page 149], delete_sec2group/1 [page 150], add_access/8 [page 150], delete_access/1 [page 150], add_view_tree_fam/4 [page 150] and delete_view_tree_fam/1 [page 151]

Own Id: OTP-4996

Reported Fixed Bugs and Malfunctions

agent Access with typo causes system crash. Adding a rudimentary type check to the (set- and is_set_ok-) access functions:

- SNMP-COMMUNITY-MIB: `snmpCommunityTable/3`
- SNMP-NOTIFICATION-MIB: `snmpNotifyTable/3`
- SNMP-TARGET-MIB: `snmpTargetAddrTable/3`, `snmpTargetParamsTable/3`
- SNMP-USER-BASED-SM-MIB: `usmUserTable/3`
- SNMP-VIEW-BASED-ACM-MIB: `vacmSecurityToGroupTable/3`, `vacmAccessTable/3` & `vacmViewTreeFamilyTable/3`

Own Id: OTP-4978
Aux Id: Seq 8380

compiler SNMP compiler cannot handle MIBs without object defs.
Luke Gorrie
Own Id: OTP-4981

agent Instrumentation function usmUserTable exited on bad values.
Own Id: OTP-3843
Aux Id: Seq 5096

Incompatibilities

-

1.13.11 SNMP Development Toolkit v3.4.2

Version 3.4.2 supports code replacement in runtime from/to version 3.4.1 and 3.4.

Improvements and new features

- Added new date and time function(s) utilizing the local_time_to_universal_time_dst of the calendar module.
  See local_time_to_date_and_time_dst [page 104] and date_and_time_to_universal_time_dst [page 101].
  The old functions, `local_time_to_date_and_time/1` and `date_and_time_to_universal_time/1`, has been obsoleted and will be removed at a later date.
  Own Id: OTP-4873

Reported Fixed Bugs and Malfunctions

- Handling of subagents (with subtrees not in sequence).
  When a subagent has two subtrees registered, A and C, and another agent has a subtree between the two, B. A get-next operation for the last variable in A would return the first variable in B, which is wrong. The master agent did check this, but not very good.
  Martin Bjrklund
  Own Id: OTP-4879

Incompatibilities

- Functions `snmp:local_time_to_date_and_time/1` and `snmp:date_and_time_to_universal_time/1`, has been obsoleted and will be removed at a later date.

1.13.12 SNMP Development Toolkit v3.4.1

Version 3.4.1 supports code replacement in runtime from/to version 3.4.

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

agent Minor errors in debug macros and sample config file.
Serge Aleynikov
Own Id: OTP-4810

agent Code up/downgrade cleanup.
Own Id: OTP-4811

compiler Fixed a parser error that caused the group checks to behave erratic. Also fixed related group check problems which among other things produced cryptic error messages.
Own Id: OTP-4825
Aux Id: Seq 8183

1.13.13 SNMP Development Toolkit v3.4

Version 3.4.0 supports code replacement in runtime from/to version 3.3.8.

Improvements and new features

agent MIB server has been re-written to improve memory usage. It is now also possible to create the “mib database” before starting the snmp agent (instead of loading mibs at runtime). MIB data is stored in either ets (default), dets or mnesia.
Own Id: OTP-4601

agent The `snmp_local_db` now use `dets` for persistent storage, instead of `snmp_pets`.
Own Id: OTP-4720

Reported Fixed Bugs and Malfunctions

compiler Added check and warning for sloppy asignment in MIBs.
Own Id: OTP-4660

compiler Fixed a bunch of errors related to group checks.
Own Id: OTP-4607
Aux Id: Seq 7765

1.13.14 SNMP Development Toolkit v3.3.8

Version 3.3.8 supports code replacement in runtime from/to version 3.3.7, 3.3.6 and 3.3.5.

Improvements and new features

- In case the UDP port dies, the snmp_net_if process now reports this and also tries to re-open the port.
  Own Id: OTP-4457
  Aux Id: Seq 7594
- SNMP mib compiler warning(s) cleanup. Some of the warnings (e.g. about missing accessfunction) was changed into info printouts, which can be seen with the compiler (erlc) argument +'{*verbosity,info*}'. See SNMP compiler options [page 98].
  Own Id: OTP-4478

Reported Fixed Bugs and Malfunctions

- The agent side set and is_set_ok operations on the snmpTargetAddrExtTable was incorrect.
  Own Id: OTP-4477
  Aux Id: Seq 7444

1.13.15 SNMP Development Toolkit v3.3.7

Version 3.3.7 supports code replacement in runtime from/to version 3.3.6, 3.3.5, 3.3.4, 3.3.3 and 3.3.2

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

- SNMP Target mib tag check incorrect.
  Own Id: OTP-4394
  Aux Id: Seq 7444

1.13.16 SNMP Development Toolkit v3.3.6

Version 3.3.6 supports code replacement in runtime from/to version 3.3.5, 3.3.4, 3.3.3 and 3.3.2

Improvements and new features

-

Reported Fixed Bugs and Malfunctions

- Improved error handling in snmp_error_report module (ets-lookup failure).
  Own Id: OTP-4345
  Aux Id: Seq 7309
- SNMP NotifiyType error. Calls to the functions `snmp_notification_mib:get_targets/0` failes since it assumes that notify type was stored as atoms, which is not always the case.
  Furthermore the parsing of the notify config file did not convert the 'trap' and 'inform' to their respective integer values 1 and 2.
  Own Id: OTP-4329
  Aux Id: Seq 7367

1.13.17 SNMP Development Toolkit v3.3.5

Version 3.3.5 supports code replacement in runtime from/to version 3.3.4, 3.3.3 and 3.3.2

Improvements and new features

- When opening a log file, the failure reason was not checked. Instead it was assumed to be `{badarg,size}` (when opened without the size option, this means the file does not exist). This is usually correct, but just to be on the safe side the test has beem changed to make sure that no other results get through.
  Own Id: OTP-4282
  Aux Id: Seq 7312
- Added possiblillity to specify own error report module (instead of the default, snmp_error). This is done with a new application config directive: `snmp_error_report_mod`, see configuration parameters [page 27]. Also added a size limit to the snmp_error module. Messages larger then 1024 chars will be truncated. Added a very simple error report module, `snmp_error_io`, which writes the message to stdout using the `io` module (without any limitations).
  Own Id: OTP-4279
  Aux Id: Seq 7309
- Test manager does not send error message in quiet mode. If the request to the manager contains an erroneous oid, no information is sent back to the client (that started the manager). See quiet config parameter [page 131] for the new reply value.
  Added two new functions for oid to/from aliasname conversion, to be used by the test manager users, see oid_to_name [page 130] and name_to_oid [page 130].
  Own Id: OTP-4250
  Aux Id: Seq 7270

Reported Fixed Bugs and Malfunctions

- Handling of large erroneous SNMP messages corrected. Encoding of the reply to these messages failed due to a bug in the length encoding. Also corrected counter increments.
  Own Id: OTP-4278
  Aux Id: Seq 7309

1.13.18 SNMP Development Toolkit v3.3.4

Version 3.3.4 supports code replacement in runtime from/to version 3.3.3, 3.3.2, 3.3.1, 3.3.0 and 3.2.2 (with the exception of what's mentioned in the version 3.3.0 note).

Improvements and new features

Reported Fixed Bugs and Malfunctions

- Crypto keys changed from string to list
  Own Id: OTP-4206
  Aux Id: Seq 7207
- SNMP date diff check changed according to RFC 2579 (was according to RFC 1903).
  Own Id: OTP-4209
  Aux Id: Seq 7185

1.13.19 SNMP Development Toolkit v3.3.3

Version 3.3.3 supports code replacement in runtime from/to version 3.3.2, 3.3.1, 3.3.0 and 3.2.2 (with the exception of what's mentioned in the version 3.3.0 note).

Improvements and new features

Reported Fixed Bugs and Malfunctions

- Erroneous macro defines corrected.
  Own Id: OTP-4006
- Storage of mib data using dets did not work (see OTP-3740).
  Own Id: OTP-4076
- Error according to section 3.2.7a of RFC 2274/2574 reported with the wrong OID (*usmStatsNotInTimeWindows* instead of *usmStatsNotInTimeWindows.0*)
  Own Id: OTP-4090

1.13.20 SNMP Development Toolkit v3.3.2

Version 3.3.2 supports code replacement in runtime from/to version 3.3.1, 3.3.0 and 3.2.2 (with the exception of what's mentioned in the version 3.3.0 note).

Improvements and new features

Reported Fixed Bugs and Malfunctions

- snmp_net_if:subtr/2 don't handle megaseconds
  Own Id: OTP-3920
  Aux Id: Seq 5174
- The mib compiler does not detect if an notification and an ordinary mib entry (OBJECT-IDENTITY) has the same OID.
  Own Id: OTP-3986
  Aux Id: Seq 5256

1.13.21 SNMP Development Toolkit v3.3.1

Version 3.3.1 supports code replacement in runtime from/to version 3.3.0 and 3.2.2 (with the exception of what's mentioned in the version 3.3.0 note).

Improvements and new features

- The UDP based Network Interface included in this application, snmp_net_if, now sets the UDP receive buffer size, according to the *snmp_net_if_recbuf* sys config option. If this option is not present, the default value is used (i.e. it is not set at all). There is no need to set the send buffer since the size of the send buffer is adjusted automatically. Note that the underlying IP implementation defines the maximum buffer size.
  Own Id: OTP-3874
  Aux Id: seq5103

Reported Fixed Bugs and Malfunctions

- Failure to retrieve mib info. This is information (specifically a list of loaded mibs) is retrieved when performing a takover. Could cause a takeover to fail.
  Own Id: OTP-3890
  Aux Id: seq5123
- Error in mib conversion for notifications. This error exist only in version 3.3.0.
  Own Id: OTP-3875
  Aux Id: seq4936
- SNMP loop if damaged snmp db. If a table row has been created with own RowIndex (key) of "", this will cause an infinit loop when traversing the table (this is done when the SNMP application at startup performs the table cleanup). This happens if:
  - Empty string for *CommunityIndex* in config file *community.conf*.
  - Empty string for *NotifyName* in config file *notify.conf*.
  - Empty string for *TargetName* in config file *target_addr.conf*.
  - Empty string for *ParamsName* in config file *target_params.conf*.

  Own Id: OTP-3881
  Aux Id: seq5113

1.13.22 SNMP Development Toolkit v3.3.0

Version 3.3.0 supports code replacement in runtime from/to version 3.2.2.

> **Note:**
> You cannot downgrade if you are using dets or mnesia for mib data storage, since previous versions only supported ets.

Improvements and new features

- The agent can now load mibs compiled with a pre 3.2.0 mib compiler.
  Own Id: OTP-3833
- Added a new interface function to retrieve the index types of the table info. This was previously internal info only. See the generic functions [page 117].
  Own Id: OTP-3816
  Aux Id: seq5053
- It is now possible to store mib data in ets, dets and mnesia. Default is ets. See configuration parameters [page 27] on how to configure this.
  Own Id: OTP-3740
  Aux Id: seq4947

Reported Fixed Bugs and Malfunctions

- The EVA application called undefined SNMP log conversion function.
  Own Id: OTP-3733
- Snmp manager 'get-bulk-request' failure.
  This is actually a UDP problem (OTP-3807). In R7 the default receive buffer, *recbuf*, size of a UDP socket has incorrectly been changed to 1024 bytes. The problem is that when a message bigger then the recbuf size is received it is cut and sizeof(recbuf) bytes is delivered (this is *not* the correct behaviour). The simple snmp manager app included in this application did not explicitly set the size of recbuf.
  So, in *R7* a get-bulk-request could easily exceed 1024 bytes, resulting in an erroneous message.
  The size of recbuf for the snmp manager app is now configurable ({`recbuf,integer()`}, see manager options [page 131]).
  Note that the maximum size of outgoing/incoming message should be set to a value less then or equal to the recbuf size! See for example *snmpEngineMaxMessageSize* in *SNMP_FRAMEWORK_MIB*.
  Note that this problem exists in R7 only!
  Own Id: OTP-3797
  Aux Id: seq5008

1.13.23 SNMP Development Toolkit v3.2.2

Version 3.2.2 supports code replacement in runtime from/to version:

- OTP version R7: 3.2.1 and 3.2.0,
- OTP version R6: 3.1.4 and 3.1.3,
- OTP version R5: 3.0.9.4, 3.0.9.3 and 3.0.9.2.

Improvements and new features

- It is now possible to register/unregister for notification of changes stored (permanetly, i.e. on disk) in snmp_local_db.
  Own Id: OTP-3704
- Added direct access (read) functions to the symbolic store for faster access (accessible throw the snmp [page 98] module).
  Own Id: OTP-3725

1.13.24 SNMP Development Toolkit v3.2.1

Version 3.2.1 supports code replacement in runtime from/to version 3.2.0, 3.1.4, 3.1.3, 3.0.9.4, 3.0.9.3 and 3.0.9.2.

Reported Fixed Bugs and Malfunctions

- Bad-arith in snmp_pdus (error case not handled). Erronous user provided messages that could not be encoded caused the application to crash.
  Own Id: OTP-3688
  Aux Id: seq4874

1.13.25 SNMP Development Toolkit v3.2.0

Version 3.2.0 supports code replacement in runtime from/to version 3.1.4, 3.1.3 and 3.0.9.2.

> **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.
>
> The required interface of the Net if module has changed [page 56].

Improvements and new features

- Debugging has been improved. It is now possible to “debug” all named processes (individually) of the snmp application. See the snmp module for details.
- Filter (audit trail) logs on timestamp.
  Own Id: OTP-3600
- The MIB-compilator has been improved. It is possible to include Description-field into compiled MIB.
  Own Id: OTP-3538

Reported Fixed Bugs and Malfunctions

- Failure converting audit trace log to text file.
  Own Id: OTP-3649, OTP-3650
  Aux Id: seq4844

1.13.26 SNMP Development Toolkit v3.1.4

Version 3.1.4 supports code replacement in runtime from/to version 3.1.3.

Improvements and new features

- Debugging has been improved. It is now possible to “debug” all named processes (individually) of the snmp application. See the snmp module for details.

Reported Fixed Bugs and Malfunctions

- Erroneous check for duplicate trap/mib entries. A check for duplicate mibentries has been added. This check can be overridden with the sys config tuple: {`snmp_mibentry_override,bool()`}. The check for duplicate trap entries was erroneous, only the first trapentry in a mib was checked. This check can now be overridden with the sys config tupple: {`snmp_trapentry_override,bool()`}. Default values in both cases are `false` (no override, which means the check is made).
  Own Id: OTP-3601
- Cloning of user from template user failure.
  Own Id: OTP-3596
  Aux Id: seq4584
- Problem with deprecated (mib-) definitions.
  A new option for the MIB-compilator is used. The option is deprecated, will get around the problem with deprecated definition.
  Own Id: OTP-3574
  Aux Id: seq4528
- Trap sending example in chapter Manual implementation corrected.
  Own Id: OTP-3353

1.13.27 SNMP Development Toolkit v3.1.3

Version 3.1.3 supports code replacement in runtime from version 3.1.2.

Improvements and new features

Reported Fixed Bugs and Malfunctions

- SNMPv3 discovery process does not work.
  Own Id: OTP-3542
  Aux Id: seq4449
- Corrupt (snmp_local_db) log files cause snmp crash. Changes to the local db is stored on disk in a logfile. In a takeover senario the new snmp will try to restore the database by reading the ets-table on disk and then update this with the transactions stored in the logfile. If the logfile is corrupt, this caused a crash.
  Own Id: OTP-3537
  Aux Id: seq4471
- Return value genErr from GET instrumentation function treated as not accepted.
  Own Id: OTP-3534
  Aux Id: seq4437
- snmp:date_and_time() rewritten to not rely on erlang:now()
  Own Id: OTP-3525
  Aux Id: seq4391
- the SNMP reportableFlag was set in response messages, which it should not.
  Own Id: OTP-3416
  Aux Id: seq4200.
- Failure to check if MIBs were already loaded at take-over.
  Own Id: OTP-3411
  Aux Id: seq4155

- Unneccessary print-outs in snmp_net_if.
  Own Id: OTP-3410
  Aux Id: seq4241
- A crash-report from disk_log was generated when the SNMP agent was started for the very first time.
  Own Id: OTP-3393
  Aux Id: seq4211
- The SNMP agent crashed (in snmp_pdus:enc_oid_tag) during initialization of table. Proper check of object identifier values has been added.
  Own Id: OTP-3378
  Aux Id: seq4155.

1.13.28 SNMP Development Toolkit v3.1.2

Version 3.1.2 supports code replacement in runtime from versions 3.1.1 and 3.0.6.

Improvements and new features

- The fact that the MIBs SNMPv2-SMI, RFC-1215, RFC-1212, SNMPv2-TC, SNMPv2-CONF and RFC1155-SMI are compiler built-ins, has been added to the compiler documentation.
  Own Id: OTP-3316
  Aux Id:
- The agent option `authentication_service` has been reintroduced. This option is part of an SNMP internal API.
  Own Id: OTP-3324
  Aux Id:
- It has been clarified in the documentation, that the value of `snmpEngineID` should not be just a simple string, but has to follow the conventions specified in RFC 2271/2571.
  Own Id: OTP-3350
  Aux Id:

Reported Fixed Bugs and Malfunctions

- If two Erlang nodes are started on the same host, and each node starts an SNMP agent, and if both agents use the same UDP port, the agent that starts last, will completely control the port. The reason for this is that the UDP port is opened with a reuse directive.
  A new option `no_reuse_address`, which, if set, causes the reuse directive not to be set.
  Own Id: OTP-3317
  Aux Id: seq4008
- Debug printouts from `snmp_net_if` appeared even when the debug flag was not set. This has been corrected.
  Own Id: OTP-3345
  Aux Id: seq4091

1.13.29 SNMP Development Toolkit v3.1.1

Improvements and new features

- The audit trail log has been improved. Now each log item also contains a time stamp. Also the text format of a log (produced by a call to `snmp:log_to_txt`) has been changed to be more "line oriented".
  The function `snmp:log_to_txt/4` has been added.
  Own Id: OTP-3261
  Aux Id: seq3884, OTP-3253
- Each item in an audit trail text log file (produced by `snmp:log_to_txt`) now has a trailing TAB character, and any TAB character in the body of a text item is replaced by ESC TAB.
  Own Id: OTP-3282
  Aux Id: seq3969
- The function `snmp:log_to_txt/5` has been added, so that not only the log name but also the log file name can be specified when converting an audit trail log to text format.
  Own Id: OTP-3298
  Aux Id:
- A new optional environment variable `bind_to_ip_addess` has been added, controlling if the agent should bind to the specific IP address or not.
  Own Id: OTP-3293
  Aux Id:

Reported Fixed Bugs and Malfunctions

- Conversion of a log to text format could crash SNMP if the log was already open.
  Own Id: OTP-3261
  Aux Id: seq3884
- The BER encoding of integers did not follow the ASN.1 BER encoding rules.
  Own Id: OTP-3274
  Aux Id: seq3960
- SNMP did not start if the audit disk_log file was corrupt.
  Own Id: OTP-3290
  Aux Id:
- SNMP was not backward compatible with instrumentation functions that returned {`noValue`, `unSpecified`} (the SNMP agent crashed). This has been changed by silently transforming such a return value to {`noValue`, `noSuchInstance`}.
  Own Id: OTP-3303
  Aux Id: seq3975
- The header file `snmp_vacm.hrl` was missing in the SNMP `src` directory.
  Own Id: OTP-3327
  Aux Id:

Incompatibilities with v3.1

- Applications that parses the audit trail log text files have to be rewritten.

1.13.30 SNMP Development Toolkit v3.1

Improvement and new features

- Adaption to new format of exit codes.

SNMP Reference Manual

Short Summaries

snmp

No functions are exported.

snmp

The following functions are exported:

- `add_agent_caps(SysORID, SysORDescr) -> SysORIndex`
  [page 98] Add an AGENT-CAPABILITY definition to the agent
- `c(File)`
  [page 98] Compile the specified MIB
- `c(File,Options) -> {ok, BinFileName} | {error, Reason}`
  [page 98] Compile the specified MIB
- `change_log_size(NewSize) -> ok | {error, Reason}`
  [page 99] Change the size of the Audit Trail Log
- `config() -> ok | {error, Reason}`
  [page 99] Configurate with a simple SNMP agent configuration tool
- `current_address() -> {value, {IP, UDP}} | false`
  [page 100] Retrieve the IP address of the manager
- `current_community() -> {value, Community} | false`
  [page 100] Retrieve the community of the current request
- `current_context() -> {value, ContextName} | false`
  [page 100] Retrieve the context of the current request
- `current_net_if_data() -> {value, NetIfData} | false`
  [page 100] Retrieve the Net_if data of the current pdu
- `current_request_id() -> {value, RequestId} | false`
  [page 100] Retrieve the request Id of the current request
- `date_and_time() -> DateAndTime`
  [page 101] Return the current date and time as an OCTET STRING
- `date_and_time_to_universal_time_dst(DateAndTime) -> [utc()]`
  [page 101] Convert a DateAndTime value to a list of possible utc()
- `date_and_time_to_string(DateAndTime) -> string()`
  [page 101] Convert a DateAndTime value to a string
- `debug(Agent,Bool) -> void()`
  [page 101] Turn debugging on/off
- `del_agent_caps(SysORIndex) -> void()`
  [page 101] Delete an AGENT-CAPABILITY definition from the agent
- `enum_to_int(Name,Enum) -> {value, Int} | false`
  [page 101] Convert an enum value to an integer
- `enum_to_int(Db,Name,Enum) -> {value, Int} | false`
  [page 102] Convert an enum value to an integer
- `get(Agent,Vars) -> Values | {error, Reason}`
  [page 102] Perform a get operation on the agent
- `get_agent_caps() -> [[SysORIndex, SysORID, SysORDescr, SysORUpTime]]`
  [page 102] Return all AGENT-CAPABILITY definitions in the agent

- send_notification(Agent,Notification,Receiver, NotifyName,ContextName,Varbinds) -> void()
  [page 106] Send a notification
- send_trap(Agent,Trap,Community)
  [page 107] Send a trap
- send_trap(Agent,Trap,Community,Varbinds) -> void()
  [page 107] Send a trap
- universal_time_to_date_and_time(UTC) -> DateAndTime
  [page 108] Convers a UTC value to DateAndTime
- unload_mibs(Agent,Mibs) -> ok | {error, Reason}
  [page 108] Unload MIBs from the agent
- unregister_subagent(Agent,SubagentOidOrPid) -> ok | {ok, SubAgentPid} | {error, Reason}
  [page 109] Unregister a subagent
- validate_date_and_time(DateAndTime) bool()
  [page 109] Check if a DateAndTime value is correct
- verbosity(Ref,Verbosity) -> void()
  [page 109] Assign a new verbosity for the process

snmp_community_mib

The following functions are exported:

- configure(ConfDir) -> void()
  [page 110] Configure the SNMP-COMMUNITY-MIB
- reconfigure(ConfDir) -> void()
  [page 110] Configure the SNMP-COMMUNITY-MIB
- add_community(Idx, CommName, SecName, CtxName, TransportTag) -> Ret
  [page 111] Added one community
- delete_community(Key) -> Ret
  [page 111] Delete one community

snmp_error

The following functions are exported:

- config_err(Format, Args) -> void()
  [page 112] Called if a configuration error occurs
- user_err(Format, Args) -> void()
  [page 112] Called if a user related error occurs

snmp_error_io

The following functions are exported:

- config_err(Format, Args) -> void()
  [page 113] Called if a configuration error occurs
- user_err(Format, Args) -> void()
  [page 113] Called if a user related error occurs

snmp_error_report

The following functions are exported:

- `config_err(Format, Args) -> void()`
  [page 114] Called if a configuration error occurs
- `user_err(Format, Args) -> void()`
  [page 114] Called if a user related error occurs

snmp_framework_mib

The following functions are exported:

- `configure(ConfDir) -> void()`
  [page 115] Configure the SNMP-FRAMEWORK-MIB
- `init() -> void()`
  [page 115] Initialize the SNMP-FRAMEWORK-MIB
- `add_context(Ctx) -> Ret`
  [page 115] Added one context
- `delete_context(Key) -> Ret`
  [page 115] Delete one context

snmp_generic

The following functions are exported:

- `get_status_col(Name,Cols)`
  [page 118] Get the value of the status column from `Cols`
- `get_status_col(NameDb,Cols) -> {ok, StatusVal} | false`
  [page 118] Get the value of the status column from `Cols`
- `get_index_types(Name)`
  [page 118] Get the index types of `Name`
- `table_func(Op1,NameDb)`
  [page 118] Default instrumentation function for tables
- `table_func(Op2,RowIndex,Cols,NameDb) -> Ret`
  [page 118] Default instrumentation function for tables
- `table_get_elements(NameDb,RowIndex,Cols) -> Values`
  [page 119] Get elements in a table row
- `table_next(NameDb,RestOid) -> RowIndex | endOfTable`
  [page 119] Find the next row in the table
- `table_row_exists(NameDb,RowIndex) -> bool()`
  [page 119] Check if a row in a table exists
- `table_set_elements(NameDb,RowIndex,Cols) -> bool()`
  [page 119] Set elements in a table row
- `variable_func(Op1,NameDb)`
  [page 119] Default instrumentation function for tables
- `variable_func(Op2,Val,NameDb) -> Ret`
  [page 119] Default instrumentation function for tables

- variable_get(NameDb) -> {value, Value} | undefined
  [page 119] Get the value of a variable
- variable_set(NameDb,NewVal) -> true | false
  [page 119] Set a value for a variable

snmp_index

The following functions are exported:

- delete(Index) -> true
  [page 122] Delete an index table
- delete(Index, Key) -> NewIndex
  [page 123] Delete an item from the index
- get(Index, KeyOid) -> {ok, {KeyOid, Value}} | undefined
  [page 123] Get the item with KeyOid
- get_last(Index) -> {ok, {KeyOid, Value}} | undefined
  [page 123] Get the last item in the index structure
- get_next(Index, KeyOid) -> {ok, {NextKeyOid, Value}} | undefined
  [page 123] Get the next item
- insert(Index, Key, Value) -> NewIndex
  [page 123] Insert an item into the index
- key_to_oid(Index, Key) -> KeyOid
  [page 123] Convert a key to an OBJECT IDENTIFIER
- new(KeyTypes)
  [page 124] Create a new snmp index structure

snmp_local_db

The following functions are exported:

- dump() -> ok | {error, Reason}
  [page 126] Dump the database to disk
- match(NameDb,Pattern)
  [page 126] Perform an ets match on the table
- print()
  [page 126] Print the database to screen
- print(TableName)
  [page 126] Print the database to screen
- print(TableName, Db)
  [page 126] Print the database to screen
- table_create(NameDb) -> bool()
  [page 126] Create a table
- table_create_row(NameDb,RowIndex,Row) -> bool()
  [page 126] Create a row in a table
- table_delete(NameDb) -> void()
  [page 127] Delete a table
- table_delete_row(NameDb,RowIndex) -> bool()
  [page 127] Delete the row in the table

- table_exists(NameDb) -> bool()
  [page 127] Check if a table exists
- table_get_row(NameDb,RowIndex) -> Row | undefined
  [page 127] Get a row from the table
- register_notify_client(Client,Module) -> ok | {error,Reason}
  [page 127] Register Client as notification client
- unregister_notify_client(Client) -> ok | {error,Reason}
  [page 127] Unregister Client as notification client

snmp_mgr

The following functions are exported:

- expect(Id, What) -> ok | {error, Id, Reason}
  [page 129] Test if the manager has received a response, trap, inform or report
- expect(Id, ErrorStatus,ErrorIndex,Varbinds)
  [page 129] Test if the manager has received a response, trap, inform or report
- expect(Id, trap, Enterp, Generic, Specific, Varbinds)
  [page 129] Test if the manager has received a response, trap, inform or report
- expect(Id, v2trap, Varbinds)
  [page 129] Test if the manager has received a response, trap, inform or report
- expect(Id, report, Varbinds)
  [page 129] Test if the manager has received a response, trap, inform or report
- expect(Id, {inform, InformReply}, Varbinds)
  [page 129] Test if the manager has received a response, trap, inform or report
- g(Oids) -> void()
  [page 129] Send a get-request
- gb(NonRepeaters, MaxRepetitions, Oids) -> void()
  [page 129] Send a get-bulk-request
- gn(Oids) -> void()
  [page 130] Send a get-next-request
- gn() -> void()
  [page 130] Send a get-next-request
- gn(N) -> void()
  [page 130] Send N get-next-request requests
- r() -> void()
  [page 130] Resend the last request
- oid_to_name(Oid) -> {ok, Name} | {error, Reason}
  [page 130] Transform a oid to it's aliasname
- name_to_oid(Name) -> {ok, Oid} | {error, Reason}
  [page 130] Transform a aliasname to it's oid
- s(Varbinds) -> void()
  [page 130] Send a set-request
- start(Options)
  [page 130] Start the SNMP manager
- start_link(Options) -> void()
  [page 131] Start the SNMP manager
- stop() -> void()
  [page 132] Stop the SNMP manager

snmp_mpd

The following functions are exported:

- `init_mpd(Options) -> mpd_state()`
  [page 133] Initialize the MPD module
- `process_packet(Packet, TDomain, TAddress, State) -> {ok, Vsn, Pdu, PduMS, ACMData} | {discarded, Reason}`
  [page 133] Process a packet received from the network
- `generate_response_msg(Vsn, RePdu, Type, ACMData) -> {ok, Packet} | {discarded, Reason}`
  [page 133] Generate a response packet to be sent to the network
- `generate_msg(Vsn, Pdu, MsgData, To) -> {ok, PacketsAndAddresses} | {discarded, Reason}`
  [page 134] Generate a request message to be sent to the network
- `discarded_pdu(Variable) -> void()`
  [page 134] Increment the variable associated with a discarded pdu

snmp_notification_mib

The following functions are exported:

- `configure(ConfDir) -> void()`
  [page 135] Configure the SNMP-NOTIFICATION-MIB
- `reconfigure(ConfDir) -> void()`
  [page 135] Configure the SNMP-NOTIFICATION-MIB
- `add_notify(Name, Tag, Type) -> Ret`
  [page 135] Added one notify definition
- `delete_notify(Key) -> Ret`
  [page 136] Delete one notify definition

snmp_pdus

The following functions are exported:

- `dec_message([byte()]) -> Message`
  [page 137] Decode an SNMP Message
- `dec_message_only([byte()]) -> Message`
  [page 137] Decode an SNMP Message, but not the data part
- `dec_pdu([byte()]) -> Pdu`
  [page 137] Decode an SNMP Pdu
- `dec_scoped_pdu([byte()]) -> ScopedPdu`
  [page 138] Decode an SNMP ScopedPdu
- `dec_scoped_pdu_data([byte()]) -> ScopedPduData`
  [page 138] Decode an SNMP ScopedPduData
- `dec_usm_security_parameters([byte()]) -> UsmSecParams`
  [page 138] Decode SNMP UsmSecurityParameters
- `enc_encrypted_scoped_pdu(EncryptedScopedPdu) -> [byte()]`
  [page 138] Encode an encrypted SNMP scopedPDU

- `enc_message(Message) -> [byte()]`
  [page 138] Encode an SNMP Message
- `enc_message_only(Message) -> [byte()]`
  [page 138] Encode an SNMP Message, but not the data part
- `enc_pdu(Pd) -> [byte()]`
  [page 138] Encode an SNMP Pdu
- `enc_scoped_pdu(ScopedPdu) -> [byte()]`
  [page 138] Encode an SNMP scopedPDU
- `enc_usm_security_parameters(UsmSecParams) -> [byte()]`
  [page 139] Encode SNMP UsmSecurityParameters

snmp_standard_mib

The following functions are exported:

- `configure(ConfDir) -> void()`
  [page 140] Configure the STANDARD-MIB and SNMPv2-MIB
- `inc(Name) -> void()`
  [page 140] Increment a variable in the MIB
- `inc(Name, N) -> void()`
  [page 140] Increment a variable in the MIB
- `reconfigure(ConfDir) -> void()`
  [page 140] Configure the STANDARD-MIB and SNMPv2-MIB
- `reinit() -> void()`
  [page 141] Reset all `snmp` counters to 0
- `sys_up_time() -> Time`
  [page 141] Get the system up time

snmp_supervisor

The following functions are exported:

- `start_sub()`
  [page 142] Start the SNMP supervisor for subagents only
- `start_sub(Opts) -> {ok, pid()} | {error, {already_started, pid()}} | {error, Reason}`
  [page 142] Start the SNMP supervisor for subagents only
- `start_master(DbDir,ConfDir)`
  [page 142] Start the SNMP supervisor for all agents
- `start_master(DbDir,ConfDir,Opts) -> {ok, pid()} | {error, {already_started, pid()}} | {error, Reason}`
  [page 142] Start the SNMP supervisor for all agents
- `start_subagent(ParentAgent,Subtree,Mibs) -> {ok, pid()} | {error, Reason}`
  [page 143] Start a subagent
- `stop_subagent(SubAgent) -> ok | no_such_child`
  [page 143] Stop a subagent

snmp_target_mib

The following functions are exported:

- `configure(ConfDir) -> void()`
  [page 144] Configure the SNMP-TARGET-MIB
- `reconfigure(ConfDir) -> void()`
  [page 144] Configure the SNMP-TARGET-MIB
- `set_target_engine_id(TargetAddrName, EngineId) -> boolean()`
  [page 145] Set the engine id for a targetAddr row.
- `add_addr(Name, Ip, Port, Timeout, Retry, TagList, Params, EngineId, TMask, MMS) -> Ret`
  [page 145] Add one target address definition
- `delete_addr(Key) -> Ret`
  [page 145] Delete one target address definition
- `add_params(Name, MPModel, SecModel, SecName, SecLevel) -> Ret`
  [page 145] Add one target parameter definition
- `delete_params(Key) -> Ret`
  [page 146] Delete one target parameter definition

snmp_user_based_sm_mib

The following functions are exported:

- `configure(ConfDir) -> void()`
  [page 147] Configure the SNMP-USER-BASED-SM-MIB
- `reconfigure(ConfDir) -> void()`
  [page 147] Configure the SNMP-USER-BASED-SM-MIB
- `add_user(EngineID, Name, SecName, Clone, AuthP, AuthKeyC, OwnAuthKeyC, PrivP, PrivKeyC, OwnPrivKeyC, Public, AuthKey, PrivKey) -> Ret`
  [page 148] Add one user
- `delete_user(Key) -> Ret`
  [page 148] Delete one user

snmp_view_based_acm_mib

The following functions are exported:

- `configure(ConfDir) -> void()`
  [page 149] Configure the SNMP-VIEW-BASED-ACM-MIB
- `reconfigure(ConfDir) -> void()`
  [page 149] Configure the SNMP-VIEW-BASED-ACM-MIB
- `add_sec2group(SecModel, SecName, GroupName) -> Ret`
  [page 150] Add one security to group definition
- `delete_sec2group(Key) -> Ret`
  [page 150] Delete one security to group definition

- `add_access(GroupName, Prefix, SecModel, SecLevel, Match, RV, WV, NV) -> Ret`
  [page 150] Add one access definition
- `delete_access(Key) -> Ret`
  [page 150] Delete one access definition
- `add_view_tree_fam(ViewIndex, SubTree, Status, Mask) -> Ret`
  [page 150] Add one view tree family definition
- `delete_view_tree_fam(Key) -> Ret`
  [page 151] Delete one view tree family definition

snmp

Application

This chapter describes the snmp application in OTP. The SNMP application provides the following services:

- a multilingual extensible SNMP agent
- a MIB compiler
- a simple manager

Configuration

The following configuration parameters are defined for the SNMP application. Refer to application(3) for more information about configuration parameters.

`audit_trail_log = false | write_log | read_write_log <optional>` Specifies if an audit trail log should be used. The `disk_log` module is used to maintain a wrap log. If `write_log` is specified, only set requests are logged. If `read_write_log`, all requests are logged. Default is `false`.

`audit_trail_log_dir = string() <optional>` Specifies where the audit trail log should be stored. If `audit_trail_log` specifies that logging should take place, this parameter must be defined.

`audit_trail_log_size = {MaxBytes, MaxFiles} <optional>` 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.

`bind_to_ip_address = bool() <optional>` If `true` the agent binds to the agent IP adress. If `false` the agent listens on any IP address on the host where it is running. Default is `false`.

`force_config_load = bool() <optional>` If `true` the configuration files are re-read during startup, 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`.

`no_reuse_address = bool() <optional>` If `true` the agent does not specify that the IP and port address should be reusable. If `false` the agent the address is set to reusable. Default is `false`.

`snmp_agent_type = master | sub <optional>` If `master`, one master agent is started. Otherwise, no agents are started. Default is `master`.

`snmp_config_dir = string() <mandatory>` Defines where the SNMP configuration files and the compiled master agent MIB files are stored.

`snmp_db_dir = string() <mandatory>` Defines where the SNMP internal db files are stored.

snmp_master_agent_mibs = [string()] <optional> Specifies a list of MIB names and defines which MIBs are initially loaded into the SNMP master agent. These MIBs are loaded from snmp_config_dir.

snmp_multi_threaded = bool() <optional> If true, the agent is multi-threaded, with one thread for each get request. Default is false.

snmp_priority = atom() <optional> Defines the Erlang priority for all SNMP processes. Default is normal.

v1 = bool() <optional> Defines if the agent shall speak SNMPv1. Default is true.

v2 = bool() <optional> Defines if the agent shall speak SNMPv2c. Default is true.

v3 = bool() <optional> Defines if the agent shall speak SNMPv3. Default is true.

snmp_local_db_auto_repair = false | true | true_verbose <optional> When starting snmp_local_db it always tries to open an existing database. If false, and some errors occur, a new datebase is created instead. If true, erroneous transactions (in the logfile) are ignored. If true_verbose, erroneous transactions (in the logfile) are igored and an error message is written. Default is true.

snmp_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 perpose of the check is to prevent that the same symbolic mibentry name is used for different oid's. Default is false.

snmp_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 perpose of the check is to prevent that the same symbolic trap name is used for different trap's. Default is false.

snmp_error_report_mod = atom() <optional> Defines an error report module, other then the default. Two modules are provided with the toolkit: snmp_error and snmp_error_io. Default is snmp_error.

snmp_master_agent_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP master agent. Default is silence.

snmp_symbolic_store_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP symbolic store. Default is silence.

snmp_note_store_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP note store. Default is silence.

snmp_net_if_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP net if. Default is silence.

snmp_mibserver_verbosity = silence | info | log | debug | trace <optional> Specifies the startup verbosity for the SNMP mib server. Default is silence.

snmp_mib_storage = ets | {dets,Dir} | {dets,Dir,Action} | {mnesia,Nodes} | {mnesia,Nod
Specifies how info retrieved from the mibs will be stored. Default is ets.
Dir = string(). Dir is the directory where the (dets) files will be created.
Nodes = [node()]. 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 table already exist.

See Also

application(3), disk_log(3)

snmp

Erlang Module

The module snmp contains interface functions to the SNMP toolkit. Some functions are off-line functions (e.g. c to compile a MIB), and some are functions called by instrumentation functions in a target system (e.g. current_address).

Common Data Types

The following datatypes are used in the functions below:

- oid() = [byte()]

The oid() type is used to represent an ASN.1 OBJECT IDENTIFIER.

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.

c(File)

c(File,Options) -> {ok, BinFileName} | {error, Reason}

Types:

- File = string()
- Options = [opt()]
- opt() = {db, volatile | persistent | mnesia} | {i, [dir()]} | {il, [dir()]} | {outdir, dir()} | {warnings, bool()} | {group_check, bool()} | {deprecated, bool()} | {description, bool()} | {verbosity, silence | warning | info | log | debug | trace}
- 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 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 warnings specifies whether warning messages should be shown. Default is true.
- 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 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 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 description specifies if the text of the DESCRIPTION field will be included or not. Default is false, in which case the description will be replaced by the atom undefined.

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 recognises the extension .mib, and invokes the SNMP MIB compiler for files with that extension. The options db, group_check and deprecated have to be specified to erlc using the syntax +term. See erlc(1) for details.

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

Types:

- NewSize = {MaxBytes, MaxFiles}
- MaxBytes = integer()
- MaxFiles = integer()

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 remebered across reboots.

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

A simple interactive SNMP agent 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 agent shall support version 3, then the crypto app must be started before running this function (password generation).

`current_address() -> {value, {IP, UDP}} | false`

Types:

- IP = [int(), int(), int(), int()]
- UDP = int()

Retrieves the IP address of the management station sending the request. It must be called from the same process that is handling the request (normally an instrumentation function).

Returns `false` if no request is currently handled.

`current_community() -> {value, Community} | false`

Types:

- Community = string()

Retrieves the community referred to in the current request. It must be called from the same process that is handling the request (normally an instrumentation function).

Returns `false` if no request is currently handled.

NOTE: This function should only be used if the agent speaks SNMPv1 or SNMPv2c only. Otherwise, use `current_context/0`.

`current_context() -> {value, ContextName} | false`

Types:

- ContextName = string()

Retrieves the context referred to in the current request. It must be called from the same process that is handling the request (normally an instrumentation function).

Returns `false` if no request is currently handled.

`current_net_if_data() -> {value, NetIfData} | false`

Types:

- NetIfData = term()

Retrieves the Net_if data for the current pdu being handled. This data is defined in the Net_if process, and can be used to forward information about the packet to the instrumentation functions. With the default Net_if implementation, it is nil. It must be called from the same process that handles the request (normally an instrumentation function).

Returns `false` if no request is currently handled.

`current_request_id() -> {value, RequestId} | false`

Types:

- RequestId = int()

Retrieves the request Id of the current request. It must be called from the same process that is handling the request (normally an instrumentation function).

Returns `false` if no request is currently handled.

`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 unversal time value on the same format as defined in calendar(3).

`date_and_time_to_string(DateAndTime) -> string()`

Types:

- DateAndTime = [int()]

Converts a DateAndTime list to a printable string, according to the DISPLAY-HINT definition in RFC1903.

`debug(Agent,Bool) -> void()`

Types:

- Agent = pid() | atom()
- Bool = bool()

Turns debugging of the agent on/off. Debug information is printed whenever an instrumentation function is called, and when a packet is received or sent. This actually sets verbosity to `log` or `silence` for the snmp_master_agent and snmp_net_if.

`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.

`enum_to_int(Name,Enum) -> {value, Int} | false`

Types:

- 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.

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. Db is a reference to the symbolic store database (retrieved by a call to get_symbolic_store_db/0<c>). <p><c>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.

get(Agent,Vars) -> Values | {error, Reason}

Types:

- Agent = pid() | atom()
- Vars = [oid()]
- 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. That the request specific parameters (such as snmp:current_request_id/0 are not accessible for the instrumentation functions if this function is used.

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_symbolic_store_db() -> Db

Types:

- Db = term()

Retrieve the symbolic store database reference. This is used for faster access to the database using the functions: int_to_enum/3, enum_to_int/3, name_to_oid/2, oid_to_name/2.

`info(Agent) -> [{Key, Value}]`

Types:

- Agent = pid() | atom()

Returns a list (a dictionary) containing information about the agent. Information includes loaded MIBs, registered subagents, some information about the memory allocation.

`int_to_enum(Name,Int) -> {value, Enum} | false`

Types:

- 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.

`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. `Db` is a reference to the symbolic store database (retrieved by a call to `get_symbolic_store_db/0<c>). <p><c>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.

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

Types:

- Mibs = [MibName]
- MibName = string()

Checks for multiple usage of object identifiers and traps between MIBs.

`load_mibs(Agent,Mibs) -> ok | {error, Reason}`

Types:

- Agent = pid() | atom()
- Mibs = [MibName]
- MibName = string()

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/",
snmp:load_mibs(snmp_master_agent, [Dir ++ "MY-MIB"]).
```

`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).

`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]
- OutFile = string()
- MibName = string()
- LogName = string()
- LogFile = string()
- Start = Stop = null | datetime() | {local_time,datetime()} | {universal_time,datetime()}

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 textfile. It defaults to `"./snmp_log.txt"`. `LogName` is the name of the log (default is `"snmp log"`), `LogFile` is the name of the log file (default is `"snmp.log"`). `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`.

`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` recognises the extension `.bin`, and invokes this function for files with that extension.

`name_to_oid(Name) -> {value, oid()} | false`

Types:

- 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.

`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. `Db` is a reference to the symbolic store database (retrieved by a call to `get_symbolic_store_db/0<c>)`. `<p><c>false` is returned if the object is not defined in any loaded MIB.

`oid_to_name(OID) -> {value, Name} | false`

Types:

- 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.

`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. `Db` is a reference to the symbolic store database (retrieved by a call to `get_symbolic_store_db/0<c>)`. `<p><c>false` is returned if the object is not defined in any loaded MIB.

`register_subagent(Agent,SubTreeOid,Subagent) -> ok | {error, Reason}`

Types:

- Agent = pid() | atom()
- SubTreeOid = oid()
- SubAgent = pid()

Registers a subagent under a subtree of another agent.

It is easy to make mistakes when registering subagents 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.

```
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()
```

Types:

- Agent = pid() | atom()
- Notification = atom()
- Receiver = no_receiver | {Tag, Recv}
- Tag = term()
- Recv = pid() | atom() | {M,F,A}
- NotifyName = string()
- ContextName = string()
- Varbinds = [Varbind]
- Varbind = {Variable, Value} | {Column, RowIndex, Value} | {OID, Value}
- Variable = atom()
- Column = atom()
- OID = oid()
- Value = term()
- RowIndex = [int()]

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. 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 acknowledgements from the managers. If the `Receiver` is specified as `no_receiver`, nothing is sent. Otherwise, it is specified as {`Tag, Recv`}. The receiver (`Recv`) gets a message:

- {`snmp_targets, Tag, Addresses`}

Addresses is a list of management target addresses. If UDP over IP is used, this is a 2-tuple {IP, UDPport}, where IP is a 4-tuple with the IP address, and UDPport is an integer. The notification is sent as an Inform-Request to each target address in Addresses. If there are no targets for which an Inform-Request is sent, Addresses is the empty list [].

For each such Address is the Addresses list, one of the following two messages is sent to Recv:

- {snmp_notification, Tag, {got_response, Address}}
- {snmp_notification, Tag, {no_response, Address}}

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.

```
send_trap(Agent,Trap,Community)
send_trap(Agent,Trap,Community,Varbinds) -> void()
```

Types:

- Agent = pid() | atom()
- Trap = atom()
- Community = string()
- Varbinds = [Varbind]
- Varbind = {Variable, Value} | {Column, RowIndex, Value} | {OID, Value}
- Variable = atom()

- Column = atom()
- OID = oid()
- Value = term()
- RowIndex = [int()]

Note! This function is only kept for backwards compatibility reasons. Use `send_notification` instead.

Sends the trap `Trap` to the managers defined for `Community` in the `intTrapDestTable` in OTP-SNMPEA-MIB. The optional argument `Varbinds` defines values for the objects in the trap. 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 trap 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 trap, 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 trap 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 trap 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, `snmp_error:user_err/2` is called and the trap is discarded.

`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 unversal time value on the same format as defined in calendar(3).

`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.

`unregister_subagent(Agent,SubagentOidOrPid) -> ok | {ok, SubAgentPid} | {error, Reason}`

Types:

- Agent = pid() | atom()
- SubTreeOidorPid = oid() | pid()

Unregisters a subagent. If the second argument is a pid, then that subagent will be unregistered from all trees in `Agent`.

`validate_date_and_time(DateAndTime) bool()`

Types:

- DateAndTime = term()

Checks if `DateAndTime` is a correct DateAndTime value, as specified in RFC1903. This function can be used in instrumentation functions to validate a DateAndTime value.

`verbosity(Ref,Verbosity) -> void()`

Types:

- Ref = pid() | snmp_master_agent | snmp_net_if | snmp_mib | snmp_symbolic_store | snmp_note_store | snmp_local_db
- 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)

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_error

Erlang Module

The module snmp_error contains two callback functions which are called by snmp_error_report 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 snmp_error_report [page 114] and configuration parameters [page 27].

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 runtime, 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)

snmp_error_io

Erlang Module

The module snmp_error_io contains two callback functions which are called by snmp_error_report 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 snmp_error_report [page 114] and configuration parameters [page 27].

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 runtime, for example if a user defined instrumentation function returns erroneous.

Format and Args are as in io:format(Format, Args).

snmp_error_report

Erlang Module

The module snmp_error_report 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 functionallity.

Two simple implementation(s) is provided with the toolkit; the modules snmp_error [page 112] which (still) is the default and module snmp_error_io [page 113].

The error report module is configured using the directive snmp_error_report_mod, see configuration parameters [page 27].

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 runtime, for example if a user defined instrumentation function returns erroneous.

Format and Args are as in io:format(Format, Args).

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()

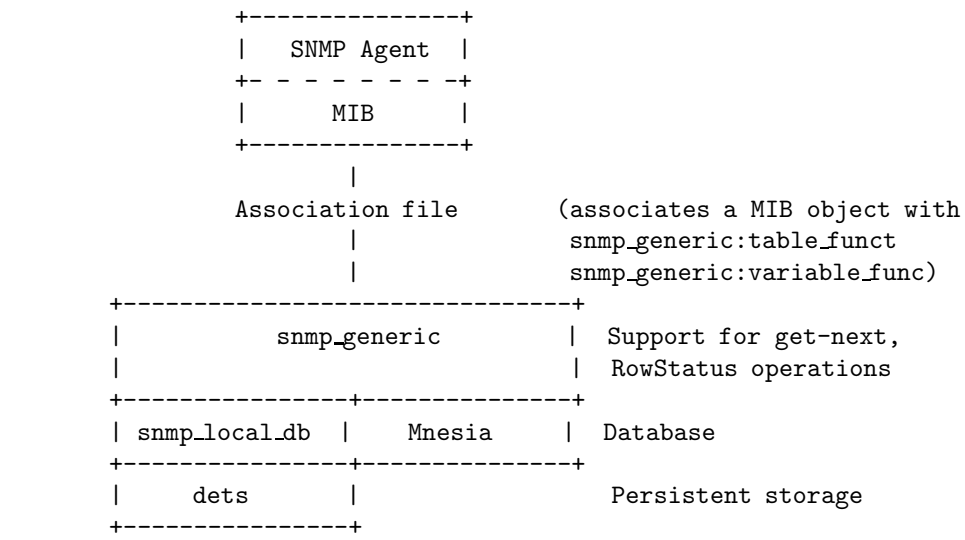
Delete a context from the agent config.

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:



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]}}.
```

In the functions defined below, the following types are used:

```
NameDb = {Name, Db}, Name = atom(), Db = volatile | persistent | mnesia
RowIndex = [int()]
Cols = [Col] | [{Col, Value}], Col = int(), Value = term()
```

`RowIndex` 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). `Cols` 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. `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

```
get_status_col(Name,Cols)
get_status_col(NameDb,Cols) -> {ok, StatusVal} | false
```

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)
```

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

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 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:

- Values = [term() | 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:

- RestOid = [int()]

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()`

Checks if a row in a table exists.

`table_set_elements(NameDb,RowIndex,Cols) -> bool()`

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

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:

- Value = term()

Gets the value of a variable.

`variable_set(NameDb,NewVal) -> true | false`

Types:

- NewVal = term()

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 [page 122] 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)`

Types:

- KeyTypes = key_types()

Creates a new snmp index structure. The `key_types()` type is described above.

snmp_local_db

Erlang Module

The module snmp_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. It is possible to manually dump the database.

There are three scaling problems with this database.

- If the database is never dumped, there are a lot of modifications to the database and the log file will grow rapidly. This can be solved by regularly dumping the database.
- The second problem occurs if the database is large, dumping the entire database may take some considerable time and it may slow down the system.
- The third problem is that insertions and deletions are inefficient for large tables.

All these problems are best solved by using Mnesia instead.

In order to know when the content of the database should be dumped, it is possible to register/unregister a notification client. This client will be notified of all persistent/permanent changes to the database by a call to:

```
Module:notify(Client,What)
```

Where the arguments are:

- `Client = term()`
- `What = insert | delete | close`

> **Note:**
> The snmp local db currently uses the defalt auto-save time of dets!

See register_notify_client/2 and unregister_notify_client/2 below for further information.

The following functions describe the interface to snmp_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 snmp_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 tha 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}`

This function can be used to dump the database at any time.

`match(NameDb,Pattern)`

Performs an ets 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 `snmp_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.

`register_notify_client(Client,Module) -> ok | {error,Reason}`

Types:

- Client = term()
- Module = atom()
- Reason = {already_registered,CurrentModule}
- CurrentModule = atom()

Register `Client` as notification client to snmp_local_db. Client is actually just used as an identity, but could e.g. be a `pid()`. When changes are made to the database (`insert`, `delete` or `stop`) notify clients will be notified.

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

Types:

- Client = term()
- Reason = not_registered

Unregister `Client` as notification client to snmp_local_db.

See Also

ets(3), snmp_generic(3)

snmp_mgr

Erlang Module

The module `snmp_mgr` provides a simple SNMP (Simple Network Management Protocol) manager. It is used for test purposes during agent development. There are two modes of operation. First, it can be used as a simple command line manager. Second, it can be used to write test suites for testing the MIB implementation in the SNMP agent.

The manager supports SNMPv1, SNMPv2c and SNMPv3, including authentication and privacy.

The command line manager uses the Erlang shell. It supports all SNMPv1, v2 and v3 requests, i.e. `set`, `get`, `get-next` and `get-bulk`. For example, `snmp_mgr:s([{[1,2,3,0], "hej"}])`, sends a set request to the agent and `snmp_mgr:g([[1,2,3,0], [myVar,0]])` gets two values. The manager operates asynchronously. This implies that the return value of most functions is nonsense. When the manager gets a response message from the agent, it is echoed to the display.

The start-up option `quiet` tells the manager not to display incoming SNMP responses, traps and informs. Messages are sent to the Erlang process that started the manager. This makes it possible to process them from an application or a test suite.

Use the `expect` function (that operates on the message queue) to write test suites. Examples of how to write a test suite can be found in `snmp_mgr_tests.erl`.

MIBs (Management Information Base) can be loaded in the manager. There are two reasons for doing this. OBJECT IDENTIFIERs (Oids) can be entered in symbolic form. Example: instead of [1,3,6,1,2,1,1,1], the symbolic name `sysDescr` can be used. The other reason is to take advantage of the type information in the MIB when sending `set` requests.

An `Oid` is represented as a list. For convenience, nested lists are allowed. There is one exception though. If an oid is entered in symbolic form, this symbol must be the first item in the list. A symbolic name includes the complete path from the top of the global naming tree. Accordingly, an oid can only contain *one* symbolic name.

Examples of valid Oids are: [myVar, 0], [1,2,3,4,5,0], [myColumn, 95], [myTable, 4, 123, [5|“eklas”]].

The last example refers to column 4 of the row with the two keys 123 and [5|“eklas”] of table myTable.

Known bug: There is not yet a {timeout, Msecs} option.

Exports

```
expect(Id, What) -> ok | {error, Id, Reason}
expect(Id, ErrorStatus,ErrorIndex,Varbinds)
expect(Id, trap, Enterp, Generic, Specific, Varbinds)
expect(Id, v2trap, Varbinds)
expect(Id, report, Varbinds)
expect(Id, {inform, InformReply}, Varbinds)
```

Types:

- Id = term()
- What = any|trap|timeout|Varbinds|ErrorStatus
- ErrorIndex = integer()
- Enterp = oid()
- Generic = integer()
- Specific = integer()
- InformReply = true | false | {error, ErrorStatus, ErrorIndex}

- `Id` is used to help identifying this particular test in a long test suite. It is not used by the manager.
- The atom `any` makes the test succeed for any response.
- `timeout` succeeds if the message queue is empty for 3.5 seconds. This can be used to ensure that no messages are pending.
- `ErrorStatus` is an atom which describes an error message. See documentation for the SNMP agent.
- `Varbinds` is a list of {Oid, Value}|{Oid,any}.

If a response other than the expected one is received, an error message is displayed and and `{error, Id, Reason}` is returned. A call to `expect` is normally directly preceeded by sending a message.

The reply to a received Inform request can be controlled. If `InformReply` is `true`, a `noError` reply is sent. If it is `false` no reply is sent. If it is `{error, ErrorStatus, ErrorIndex}`, a reply indicating the error is sent.

```
g(Oids) -> void()
```

Types:

- Oids = [oid()]

Sends a `get-request`.

```
gb(NonRepeaters, MaxRepetitions, Oids) -> void()
```

Types:

- NonRepeaters = integer()
- MaxRepetitions = integer()
- Oids = [oid()]

Sends a `get-bulk-request` (See RFC1905).

`gn(Oids) -> void()`

Types:

- Oids = [oid()]

Sends a `get-next-request`.

`gn() -> void()`

Sends yet another get-next-request constructed from the previous response. This is a nice feature for manually traversing a MIB.

`gn(N) -> void()`

Types:

- N = integer()

Sends N `get-next-request` requests.

The last response is used as the start value. Works somewhat like a `get-bulk-request` (see SNMPv2).

`r() -> void()`

Resend the last request.

`oid_to_name(Oid) -> {ok, Name} | {error, Reason}`

Types:

- Oid = oid()
- Name = atom()

Transform a oid to it's aliasname.

`name_to_oid(Name) -> {ok, Oid} | {error, Reason}`

Types:

- Name = atom()
- Oid = oid()

Transform a aliasname to it's oid.

`s(Varbinds) -> void()`

Types:

- Varbinds = [varbind()]

Sends a `set-request`.

Varbind is:

- {Oid, Value} if the object with Oid `Oid` is loaded by the manager.
- {Oid, TypeTag, Value} where TypeTag is `s|o|i` (String, Oid, Integer). This syntax is used if this object is not defined in a MIB loaded by the manager. (Or if you explicitly want to send a request of wrongly typed data.)

`start(Options)`

`start_link(Options) -> void()`

Types:

- Options = [options()]

Starts the SNMP manager.

Mandatary options are:

- {agent, Agent} - where Agent is the IP address of the agent {int(),int(),int(),int()} or the name of the host (string()).

Optional options are:

- {`agent_udp, int()`} - the UDP port that the agent listens to. Default is 4000.
- {`trap_udp, int()`} - the UDP port where the manager will receive traps. Default is 5000.
- {`community, string()`} - the community string that is sent in the requests from the manager. Default is “public”.
- {`context, string()`} - the context that is sent in v3 requests from the manager. Default is “”.
- {`user, string()`} - the USM user name that is sent in v3 requests from the manager. Default is “initial”.
- {`engine_id, string()`} - the engine ID of the agent. Used in v3 only. Default is “agentEngine”.
- {`context_engine_id, string()`} - the context engine ID used in v3 requests. Default is the same as `engine_id`.
- {`sec_level, noAuthNoPriv + authNoPriv | authPriv`} - the requested security level. Used in v3 only. Default is `noAuthNoPriv`.
- {`dir, string()`} - the directory where the file `usm.conf` is located. This file is only needed if v3 is used. The file has the same syntax as the usm.conf file for the agent.
- {`mibs, List of filename`} - MIBs to be loaded in the manager. Default is no MIBs. The MIBs must be compiled.
- {`receive_type, pdu | msg`} - defines the format of delivered messages. Default is `pdu`.
- `quiet` - incoming responses are not displayed.
  Messages are sent to the Erlang process that started the manager. The format of the message depends on the value of `receive_type`. If the value is `pdu` (default),the message is {snmp_pdu, PDU} where PDU is a pdu() or a trappdu() record defined in `snmp_types.hrl`. If the value is `msg`, the message is {`snmp_msg, Msg, Ip, Udp`}. If the request was issued with an erroneous oid, the message is {oid_error, Reason}, where Reason is a printable string. Default is, this option is not present, i.e. all incoming requests are displayed. This option must be present when running test suites.
- `v1|v2|v3` - what SNMP version to use. Default is v1.
- {`recbuf, integer()`} - defines the size of a UDP socket receive buffer.
  This is important when sending large regusts to the agent (i.e. requests which will gererate large responses).
  Also consider the max size of the agents outgoing message (defined e.g. by *snmpEngineMaxMessageSize* in SNMP-FRAMEWORK-MIB). Default is `1024`.

```
stop() -> void()
```

Stops the SNMP manager.

snmp_mpd

Erlang Module

The module snmp_mpd implements the version independent Message Processing and Dispatch functionality in SNMP. It is supposed to be used from a Network Interface process (net_if).

Exports

`init_mpd(Options) -> mpd_state()`

Types:

- Options = [Option]
- Option = v1 | v2 | v3

This function can be called from the net_if process at startup. The options list defines which versions to use.

It also initializes some SNMP counters.

`process_packet(Packet, TDomain, TAddress, State) -> {ok, Vsn, Pdu, PduMS, ACMData} | {discarded, Reason}`

Types:

- Packet = binary()
- TDomain = snmpUDPDomain
- TAddress = {Ip, Udp}
- Ip = {integer(), integer(), integer(), integer()}
- Udp = integer()
- State = mpd_state()
- Vsn = 'version-1' | 'version-2' | 'version-3'
- Pdu = #pdu
- PduMs = integer()
- ACMData = acm_data()

Processes an incoming packet. Performs authentication and decryption as necessary. The return values should be passed the agent.

`generate_response_msg(Vsn, RePdu, Type, ACMData) -> {ok, Packet} | {discarded, Reason}`

Types:

- Vsn = 'version-1' | 'version-2' | 'version-3'
- RePdu = #pdu

- Type = atom()
- ACMData = acm_data()
- Packet = binary()

Generates a possibly encrypted response packet to be sent to the network. `Type` is the `#pdu.type` of the original request.

`generate_msg(Vsn, Pdu, MsgData, To) -> {ok, PacketsAndAddresses} | {discarded, Reason}`

Types:

- Vsn = 'version-1' | 'version-2' | 'version-3'
- Pdu = #pdu
- MsgData = msg_data()
- 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.

`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.

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 de 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`.

`reinit() -> 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_supervisor

Erlang Module

The snmp_supervisor is the supervisor for the SNMP application. There is always one supervisor at each node with an SNMP agent (master agent or subagent).

Exports

start_sub()

start_sub(Opts) -> {ok, pid()} | {error, {already_started, pid()}} | {error, Reason}

Types:

- Opts = [opt()]
- opt() = {priority, Prio}

Starts a supervisor for the SNMP agent system without a master agent. The supervisor starts all involved SNMP processes, but no agent processes. Subagents should be started by calling start_subagent/3.

Prio is an Erlang priority. All SNMP processes use this priority. Default is the same as default in the Erlang runtime system.

start_master(DbDir,ConfDir)

start_master(DbDir,ConfDir,Opts) -> {ok, pid()} | {error, {already_started, pid()}} | {error, Reason}

Types:

- DbDir = string()
- ConfDir = string()
- Opts = [opt()]
- opt() = {mibs, Mibs} | {net_if, NetIfModule} | {priority, Prio} | {name, Name}
- Mibs = [MibName]
- MibName = [string()]
- NetfModule = atom()
- Name = {local, atom()} | {global, atom()}

Starts a supervisor for the SNMP agent system. The supervisor starts all involved SNMP processes, including the master agent. Subagents should be started by calling `start_subagent/3`.

`DbDir` is a string including a trailing directory delimiter, which points to the directory where the database files sre stored.

`ConfDir` is a string including a trailing directory delimiter, which points to the directory where the configuration file is found.

If the STANDARD-MIB is not specified in the `Mibs` list, it is loaded from the configuration directory (i.e. with the `.conf` files).

If no `NetIfModules` is specified, the default net if implementation is used (`snmp_net_if`).

`Prio` is an Erlang priority. All SNMP processes use this priority. Default is the same as default in the Erlang runtime system.

If no `Opts` is given, `[{name, {local, snmp_master_agent}}]` is default.

`start_subagent(ParentAgent,Subtree,Mibs) -> {ok, pid()} | {error, Reason}`

Types:

- ParentAgent = pid()
- SubTree = oid()
- Mibs = [MibName]
- MibName = [string()]

Starts a subagent on the node where the function is called. The `snmp_supervisor` must be running.

If the supervisor is not running, the function fails with the reason `badarg`.

`stop_subagent(SubAgent) -> ok | no_such_child`

Types:

- SubAgent = pid()

Stops the subagent on the node where the function is called. The `snmp_supervisor` must be running.

If the supervisor is not running, the function fails with the reason `badarg`.

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 enigne 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.

List of Figures

List of Tables

Index of Modules and Functions

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