

# **Simple Network Management Protocol (SNMP)**

**version 3.0**

**Martin Björklund, Klas Eriksson**

**1998-04-27**

Typeset in  $\text{\LaTeX}$  from SGML source using the DOCBUILDER 3.0 Document System.



# Contents

<b>1</b>	<b>SNMP User's Guide</b>	<b>1</b>
1.1	SNMP Introduction . . . . .	2
	Scope and Purpose . . . . .	2
	Prerequisites . . . . .	2
	About This Manual . . . . .	2
	Where to Find More Information . . . . .	3
1.2	Functional Description . . . . .	4
	Definitions . . . . .	4
	Features . . . . .	5
	SNMPv1, SNMPv2 and SNMPv3 . . . . .	5
	Operation . . . . .	6
	Subagents and MIB Loading . . . . .	9
	Contexts and Communities . . . . .	9
	Management of the Agent . . . . .	10
	Notifications . . . . .	15
1.3	Instrumentation Functions . . . . .	18
	Instrumentation Functions . . . . .	18
	Using the ExtraArgument . . . . .	23
	Default Instrumentation . . . . .	24
	Atomic Set . . . . .	24
1.4	The MIB Compiler . . . . .	26
	Operation . . . . .	26
	Importing MIBs . . . . .	26
	MIB Consistency Checking . . . . .	27
	.hrl File Generation . . . . .	27
	Emacs Integration . . . . .	27
	Compiling from a shell or a Makefile . . . . .	28
	Deviations from the standard . . . . .	28
1.5	Running the Agent . . . . .	29
	Configuring the Agent . . . . .	29
	Modifying the Configuration Files . . . . .	30

	Starting the Agent . . . . .	31
	Debugging the Agent . . . . .	31
1.6	Implementation Example . . . . .	32
	MIB . . . . .	32
	Default Implementation . . . . .	34
	Manual Implementation . . . . .	35
1.7	Advanced Topics . . . . .	41
	When to use a Subagent . . . . .	41
	Agent Semantics . . . . .	42
	Subagents and Dependencies . . . . .	42
	Distributed Tables . . . . .	43
	Fault Tolerance . . . . .	43
	Using Mnesia Tables as SNMP Tables . . . . .	44
	Audit Trail Logging . . . . .	47
	Deviations from the standard . . . . .	47
1.8	Definition of Configuration Files . . . . .	48
	Agent Information . . . . .	48
	Contexts . . . . .	48
	System Information . . . . .	49
	Communities . . . . .	49
	MIB Views for VACM . . . . .	50
	Security data for USM . . . . .	50
	Trap Destinations . . . . .	51
	Notify Definitions . . . . .	51
	Target Address Definitions . . . . .	51
	Target Parameters Definitions . . . . .	52
1.9	Definition of Instrumentation Functions . . . . .	53
	Variable Instrumentation . . . . .	53
	Table Instrumentation . . . . .	55
1.10	Definition of Net if . . . . .	58
	Mandatory Functions . . . . .	58
	Messages . . . . .	59
1.11	SNMP Appendix A . . . . .	62
	Appendix A . . . . .	62
1.12	SNMP Appendix B . . . . .	63
	Appendix B . . . . .	63
1.13	SNMP Release Notes . . . . .	72
	SNMP Development Toolkit v3.0.1 . . . . .	72
	SNMP Development Toolkit v3.0 . . . . .	72
	SNMP Development Toolkit v2.2.3 . . . . .	73

SNMP Development Toolkit v2.2.2 . . . . .	73
SNMP Development Toolkit v2.2.1 . . . . .	74
SNMP Development Toolkit v2.2 . . . . .	74
SNMP Development Toolkit v2.1.1 . . . . .	76
SNMP Development Toolkit v2.1.1 . . . . .	76
SNMP Development Toolkit v2.1 . . . . .	77
SNMP Development Toolkit v2.0 . . . . .	77
SNMP Development Toolkit v1.3.1 . . . . .	79
SNMP Development Toolkit v1.3 . . . . .	79
SNMP Development Toolkit v1.2.1 . . . . .	80
SNMP Development Toolkit v1.2 . . . . .	81
<b>2 SNMP Reference Manual</b>	<b>83</b>
2.1 snmp (Application) . . . . .	92
2.2 snmp (Module) . . . . .	94
2.3 snmp_community_mib (Module) . . . . .	104
2.4 snmp_error (Module) . . . . .	105
2.5 snmp_framework_mib (Module) . . . . .	106
2.6 snmp_generic (Module) . . . . .	107
2.7 snmp_index (Module) . . . . .	111
2.8 snmp_local_db (Module) . . . . .	115
2.9 snmp_mgr (Module) . . . . .	118
2.10 snmp_mpd (Module) . . . . .	122
2.11 snmp_notification_mib (Module) . . . . .	124
2.12 snmp_pdus (Module) . . . . .	125
2.13 snmp_standard_mib (Module) . . . . .	128
2.14 snmp_supervisor (Module) . . . . .	130
2.15 snmp_target_mib (Module) . . . . .	132
2.16 snmp_user_based_sm_mib (Module) . . . . .	133
2.17 snmp_view_based_acm_mib (Module) . . . . .	134
<b>List of Figures</b>	<b>137</b>
<b>List of Tables</b>	<b>139</b>



# 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 tedious `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.

## Scope and Purpose

This manual describes the SNMP development tool, version 3.0, 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.

## Prerequisites

The following prerequisites knowledge is required for understanding the material in this 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.

## About This Manual

In addition to this introductory chapter, this 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: "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 more detail.
- Appendix A describes the conversion of SNMPv2 to SNMPv1 error messages.
- Appendix B contains the RFC1903 text on RowStatus.

## 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
- 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. 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 that can be used for simple interactive testing, and for writing test suites.

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

This 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.

## Definitions

The following definitions will help users understand the material presented in this user's guide.

**MIB** The conceptual repository for management information is called the Management Information Base (MIB). It is conceptual because 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 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.

## 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.

## SNMPv1, SNMPv2 and SNMPv3

Currently (fall 1998), SNMPv1 is the full internet standard to use for network management. However, the “new” set of SNMPv2 specifications (RFC1902-1907) are now draft internet standards. Many managers choose to support both of these protocols. 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 what 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. There are other experimental frameworks as well, e.g. SNMPv2u and SNMPv2\*.

The SNMPv3 specifications (RFC2271-2275), which further enhances SNMPv2, are now draft internet standards as well. These 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 can't 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 RFC1908 and RFC2089.

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.

The translation from a SMIV1 MIB to a 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 a SNMPv1 manager can't decode correctly. Therefore, an agent may never send a Counter64 object to a SNMPv1 manager. The common practice in these situations is to simply ignore any Counter64 objects, when sending a reply or a trap to a SNMPv1 manager. For example, if a SNMPv1 manager tries to GET an object of type Counter64, he will get a noSuchName error, while a SNMPv2 manager would get a correct value.

## 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.

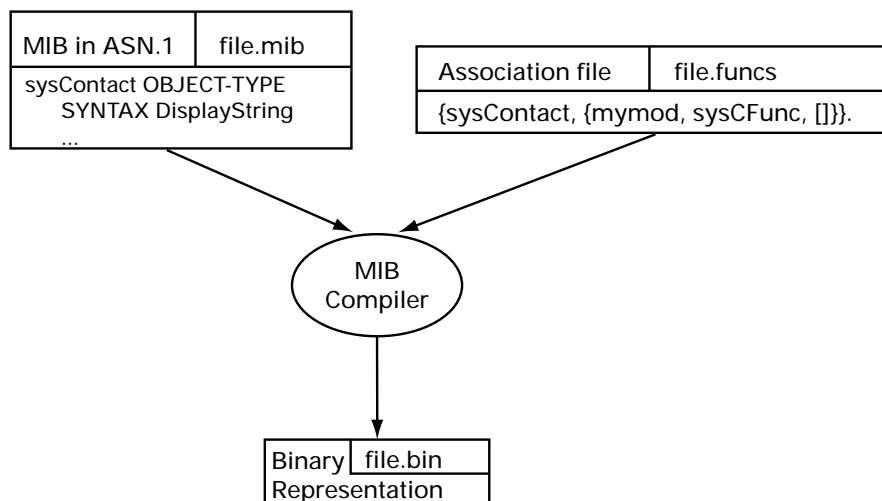


Figure 1.1: MIB Compiler Principles

The compiler parses the SMI file and associates each table or variable with an instrumentation function (see MIB Compiler Principles [page 7]). 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 Starting the Agent [page 7]). The instrumentation is ordinary Erlang code which is loaded explicitly or automatically the first time it is called.

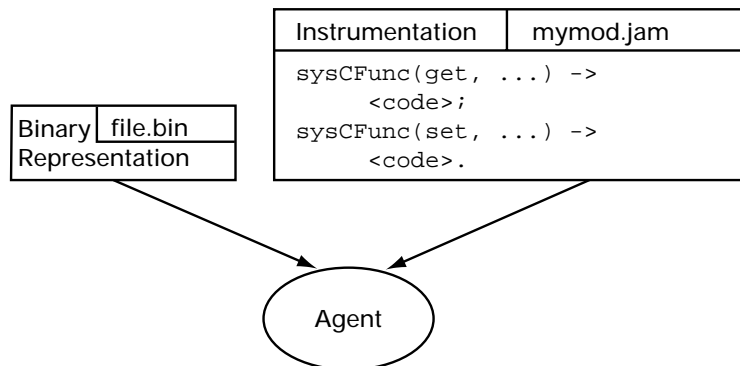


Figure 1.2: Starting the Agent

The SNMP agent system consists of one Master Agent and optional subagents. The Master Agent can be thought of 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.

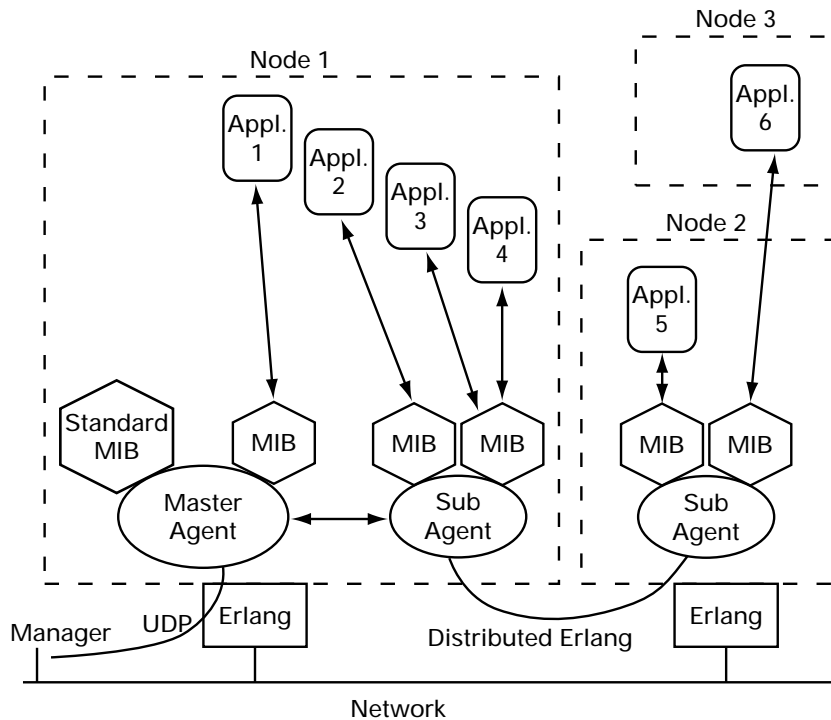


Figure 1.3: Architecture

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. Every 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.

## 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 cards 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 registers and de-registers itself. A subagent can register itself below 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. We do not need 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 41] 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.

## 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 an SNMP entity. The method for identifying instances specified by the MIB module does not allow each instance to be distinguished amongst the set of all instances within an SNMP entity; rather, it allows each instance to be identified only within some scope or "context", where there are multiple such contexts within the SNMP entity. Often, a context is a physical device, or perhaps a logical device, although a context can also encompass multiple devices, or a subset of a single device, or even a subset of multiple devices, but a context is always defined as a subset of a single SNMP entity. Thus, in order to identify an individual item of management information within an SNMP entity, the context must be identified in addition to its object type and its instance.



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, a SNMPv1, SNMPv2c or a 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 29].

## Management of the Agent

There are 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 that the data that the MIBs define is used internally in the agent, even if the MIBs aren't 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. Provided in the distribution are MIBs and implementations for both of these versions. The MIB file for SNMPv1 is called STANDARD-MIB, and the corresponding for SNMPv2 is called SNMPv2-MIB. If the agent is configured for SNMPv1 only, the STANDARD-MIB is loaded by default; otherwise, the SNMPv2-MIB is loaded by default. It is possible to override this default behavior, by explicitly loading another version of this MIB, for example, you could choose to implement the union of all objects in these two MIBs.

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

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

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

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

The ASN.1 code, the Erlang source code, and the generated `.hrl` files for them are provided in the distribution, 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 29].

## 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 you need another implementation of any of these objects, e.g. you may want to store the persistent variables in the Mnesia database instead, you will have to make your own implementation of these variables (you can of course use the standard implementation for the other variables), compile and load the MIB 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 29].)

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

These MIBs 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

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

**snmpTargetAddrTable** An entry in this table 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 this table. Each row in the `snmpTargetAddrTable` refers to an entry in the `snmpTargetParamsTable`.

**snmpTargetParamsTable** An entry in this table 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.



**vacmAccessTable** This table 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** This table 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**

This 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**

This 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 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.

**OTP-SNMPEA-MIB**

This 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.

**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,Varbinds,ContextName])
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 is to call the function as `snmp:send_notification(Agent, Notification, no_receiver)`. In this case, the agent performs a get-operation to retrieve the object values which 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 `snmp_error:user_err/2`. 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 which 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 calls to `get-next`.

### 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 \[page 53\]](#).

#### 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 loaded and unloaded, 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.

## 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.

## GetNext Operation

This 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 1.4: Contents of `my_table`

**Note:**

N/A means not accessible.

The manager issues the following `getNext` request:

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

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

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

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

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

This is illustrated in the following table:

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

Figure 1.5: `GetNext` from `[3,1,1]` and `[5,1,1]`.

The manager now issues the following `getNext` request:

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

This is transformed into one call to `my_table`:

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

The function should now return:

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

This is illustrated in the following table:

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

endOfTable

Figure 1.6: `GetNext` from `[3,2,1]` and `[5,2,1]`.

The manager now issues the following getNext request:

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

This will be transform into one call to my\_table:

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

The function should now return:

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

This is illustrated in the following table:

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

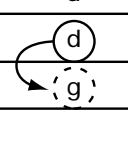


Figure 1.7: GetNext from [3,1,2] and [4,1,2].

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.

## 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]}}.
```

## 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.

## 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.

This part of phase one is handled internally by the agent. 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

This section describes the MIB compiler and contains the following topics:

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

### Operation

The MIB must be written as a text file in SMIV1 or SMIV2 before being 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 see Default Instrumentation [page 24] 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 determine if the MIB is written in SMI version 1 or 2.

### Importing MIBs

The compiler handles the `IMPORT` statement. It is the compiled file which is imported, not the ASN.1 file. Therefore, an MIB must be recompiled to make changes visible to other MIBs which import 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 latest version of the `snmp` and `myapp` applications in the OTP system and uses the expanded paths as include paths.

Note that a SMIV2 MIB can import an SMIV1 MIB and vice versa.

## MIB Consistency Checking

When an MIB is compiled, the compiler detects if several managed objects use the same OBJECT IDENTIFIER. If this is the case, it issues an error message. However, the compiler cannot detect Oid conflicts between different MIBs. These kind 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"]`. This function also performs consistency checking on trap definitions.

## .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).
```

## 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`.

## 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
```

## Deviations from the standard

In some aspects the Erlang MIB compiler doesn't 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.

## 1.5 Running the Agent

This chapter 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 48] which contains more detailed information about the configuration files.

### 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 48] 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. These parameters should be defined in an Erlang system configuration file. The following configuration parameters are defined for the SNMP application: <! NOTE: This list is duplicated from snmp\_app.sgml ->

`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.

`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`.

`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.

## Modifying the Configuration Files

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 won't 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 "res"  
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

## 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 won't start, but report a `{config_error, {unsupported_crypto, _}}` error.

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

ok

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

ok

## Debugging the Agent

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 instrumentation functions.

```
3> snmp:debug(snmp_master_agent, true).
```

ok

```
4>
```

```
%% Example of output from the agent when a get-request arrives:
```

```
** SNMP Agent debug: got pdu from {{147,12,12,12},5000},
    community "public"
```

```
    pdu: {pdu,'get-next-request',31123857,noError,0,
    [{varbind,[1,1,2],'NULL','NULL',1]}}
```

```
** SNMP Agent debug:
```

```
    apply: snmp_generic_variable_func,[get,{sysDescr,permanent}]
    returned: {value,"Erlang SNMP agent"}
```

```
** SNMP Net if debug:
```

```
    reply pdu: {pdu,'get-response',31123857,noError,0,
    [{varbind,[1,3,6,1,2,1,1,1,0],'OCTET STRING',
    "Erlang SNMP agent",1]}}
```

```
4>
```

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

This section describes how an MIB can be implemented with the SNMP Development Toolkit. The example shown can be found in the toolkit distribution.

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

### 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



## 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"},
{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>
```

## Manual Implementation

The following example shows a “manual” implementation of this MIB in Erlang. In this example, the values of the objects are stored in an Erlang server. This 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:

This is an example only, and not the best way to implement a table manually. Use for example the module `snmp_index` to implement an efficient table.

### 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 doesn't 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

```

```
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(snmpp).
ok
2> snmp:c("EX1-MIB").
{ok,"EX1-MIB.bin"}
3> snmp:load_mibs(snmpp_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

This section shows how to send a trap by sending the `fTrap` from the master agent. 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).

In this example, 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-a
{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(MAPid, 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

This chapter 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

### When to use a Subagent

This section 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.

## 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 the atomicity of `set` requests. If this is hard to do, don't 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 on one occasion. You cannot even assume that requests which belong 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 for this variable. If called once only, there is no definition which determines which of the new values will 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`.

## 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.

## 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. It is responsible for coordinating the processes which hold the table data and is called a table holder. All table holders must 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, then a very specialized table coordinator could be implemented.

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.

## 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 won't work.

One solution to this problem is to make the snmp application a distributed Erlang application. This means that 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 node, and this may be a problem for the SNMP managers communicating with 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, you will have to load the MIBs explicitly after takeover.

## 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 used 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 which are used internally, but should not be visible to a manager. These internal columns must be the last columns in the table. Also, the set operation will not work with this arrangement because there are columns which 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 now give a value to the internal column. The instrumentation functions now looks 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".

## 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.

## Deviations from the standard

In some aspects the agent doesn't 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 doesn't 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 a 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 which 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 10] for a description of these 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 `snmp_error:config_err/2` at start-up.

### Agent Information

This information should be stored in a file called `agent.conf`, which must be present.

Each entry is a tuple of size two:

`{AgentVariable, Value}.`

- `AgentVariable` is one of the variables in 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 valid `agent.conf` file:

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

### Contexts

This information should be stored in a file called `context.conf`, which must be present. 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.

## System Information

This information should be stored in a file called `standard.conf`, which must be present.

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.

## Communities

This 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 string.
- CommunityName is a string.
- SecurityName is a string.
- ContextName is a string.
- TransportTag is a string.



## MIB Views for VACM

This information should be stored in a file called `vacm.conf`, which must be present.

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<c>`, or `<c>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.

## Security data for USM

This information 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 `usmNoAuthProtocol`, `usmHMACMD5AuthProtocol`, or `usmHMACSHAAuthProtocol`.
- `AuthKeyC` is a string.
- `OwnAuthKeyC` is a string.

- PrivP is a `usmNoPrivProtocol<c>` or `<c>usmDESPrivProtocol`.
- PrivKeyC is a string.
- OwnPrivKeyC is a string.
- Public is a string.
- AuthKey is a string. This is the User's secret localized authentication key. It is not visible in the MIB.
- PrivKey is a string. This is the User's secret localized encryption key. It is not visible in the MIB.

## Trap Destinations

This information was previously stored in a file called `trap_dest.conf`. If the agent encounters this file, but not the new target configuration files (see below), this file is automatically converted to the new files, and these new files are read.

## Notify Definitions

This information should be stored in a file called `notify.conf`, which must be present.

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

Each entry is a term:

`{NotifyName, Tag, Type}`.

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

## Target Address Definitions

This information should be stored in a file called `target_addr.conf`, which must be present.

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, TMask, MaxMessageSize}`.

- TargetName is a unique 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.
- TMask is a string of size 0, or size 6.
- MaxMessageSize is an integer.

## Target Parameters Definitions

This information 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 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

This section describes the user defined functions which the agent calls at different times.

### 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. We recommend that you use the SNMPv2 error codes for all instrumentation functions, as these provide more details. See Appendix A [page 62] for a description of error code conversions.

#### **f(new [, ExtraArgs])**

This function 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])**

This function 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])**

This function is called when a get-request or a get-next request refers to the variable.

This function is mandatory.

#### Valid Return Values

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

### **f(is\_set\_ok, NewValue [, ExtraArgs])**

This function 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)`

## 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. We recommend that you use the SNMPv2 error codes for all instrumentation functions, as these provide more details. See Appendix A [page 62] for a description of error code conversions.

### **f(new [, ExtraArgs])**

This function 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])**

This function 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])**

This function is called when a get-request refers to a table.

This function is mandatory.

### Arguments

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

### Valid Return Values

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

- `genErr`. Used if an error occurred. Note that this should be an internal processing error, e.g. a caused by a programming fault somewhere. If some column does not exist, use `{noValue, noSuchName}` or `{noValue, noSuchInstance}`.

### **f(get\_next, RowIndex, Cols [, ExtraArgs])**

This function is called when a get-next- or a get-bulk-request refers to the table.

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

This function is mandatory.

#### **Arguments**

- `RowIndex` is a list of integers (possibly empty) which 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 IDENTIFIER 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 programming fault somewhere. If some column does not exist, you must return the next accessible element (or `endOfTable`).

### **f(is\_set\_ok, RowIndex, Cols [, ExtraArgs])**

This function is called in phase one of the set-request processing so that new values can be checked for inconsistencies.

If this 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, this function 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])**

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 did not exist, or returned {noError, 0}.

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 {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

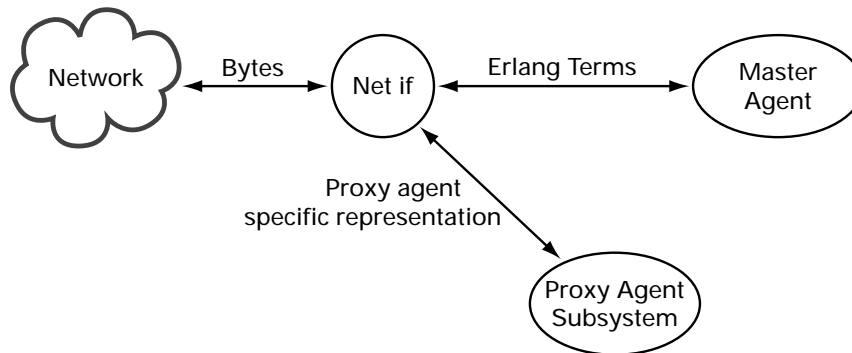


Figure 1.8: The Purpose of Net if

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 it 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, this 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 the Net if process can 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.

### Mandatory Functions

A Net if process must be implemented in a module which exports the `Module:start_link/1` function. This function 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)`

#### Arguments

`MasterAgent` is a `Pid`.

## 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.

## Messages

This section 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 which is aimed for the MasterAgent:

MasterAgent ! {snmp\_pdu, Vsn, Pdu, PduMS, ACMDData, 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 further below).
- ACMDData is data used by the Access Control Module in use. Normally this is returned from `snmp_mpd:process_packet` (see further below).
- From is the source address. If UDP over IP is used, this should be a 2-tuple {IP, UDPport}, where IP is a 4-tuple with the IP address, and UDPport is an integer.
- Extra is any term the Net if process wishes to send to the agent. This term can be retrieved by the instrumentation functions by calling `snmp:current_net_if_data()`. This data is also sent back to the Net if process when the agent generates a response to the request.

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

Pid ! {snmp\_response\_received, Vsn, Pdu, From}

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

## Incoming Messages

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

- {`snmp_response`, `Vsn`, `Pdu`, `Type`, `ACMData`, `To`, `Extra`} This message is sent to the Net if process from a master agent as a response to a previously received request.
  - `Vsn` is either 'version-1', 'version-2', or 'version-3'.
  - `Pdu` is an SNMP PDU record (as defined in `snmp_types.hrl`) with the SNMP response.
  - `Type` is the `#pdu.type` of the original request.
  - `ACMData` is data used by the Access Control Module in use. Normally this is just sent to `snmp_mpd:generate_response_message` (see further below).
  - `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 further below).
  - `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

### Appendix A

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

Mapping of SNMPv2 error message to SNMPv1:

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

Table 1.1: -

## 1.12 SNMP Appendix B

### 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				
ACTION	A	B	C	D
	status column does not exist	status col. is notReady	status column is notInService	status column is active
set status	noError	->D	inconsistent-	inconsistent-
column to	or	entValue	Value	Value
createAndGo	inconsistent-			
	Value			
set status	noError	see 1	inconsistent-	inconsistent-
column to	or	entValue	Value	Value

createAndWait	wrongValue				
-----	-----	-----	-----	-----	
set status	inconsistent-	inconsist-	noError	noError	
column to	Value	entValue			
active		or			
		see 2 ->D		->D	->D
-----	-----	-----	-----	-----	
set status	inconsistent-	inconsist-	noError	noError	->C
column to	Value	entValue			
notInService		or		or	
		see 3 ->C		->C wrongValue	
-----	-----	-----	-----	-----	
set status	noError	noError	noError	noError	
column to					
destroy		->A	->A	->A	->A
-----	-----	-----	-----	-----	
set any other	see 4	noError	noError	see 5	
column to some					
value		see 1		->C	->D
-----	-----	-----	-----	-----	

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

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

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

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

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

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

noError: because the agent chooses to create the instance.

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



`inconsistentName` or `inconsistentValue` is returned, the row remains in state A.

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

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

### Conceptual Row Creation

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

#### Interaction 1: Selecting an Instance-Identifier

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

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

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

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

management station should simply select a new pseudo-random number and retry the operation.

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

#### Interaction 2: Creating the Conceptual Row

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

#### Interaction 2a: Creating and Activating the Conceptual Row

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

- a value is returned, indicating that some other management station has already created this conceptual row. We return to interaction 1.
- the exception 'noSuchInstance' is returned, indicating that the agent implements the object-type associated with this column, and that this column in at least one conceptual row would be accessible in the MIB view used by the retrieval were it to exist. For those columns to which the agent provides read-create access, the 'noSuchInstance' exception tells the management station that it should supply a value for this column when the conceptual row is to be created.
- the exception 'noSuchObject' is returned, indicating that the agent does not implement the object-type associated with this column or that there is no conceptual row for which this column would be accessible in the MIB view used by the retrieval. As such, the management station can not 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 can not 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

### SNMP Development Toolkit v3.0.1

#### Improvements and new features

- A new function `snmp:change_log_size/1` has been added to dynamically change the size of the Audit Trail Log.  
Own Id: OTP-2989  
Aux Id: seq1499

#### Reported Fixed Bugs and Malfunctions

- The function `snmp:get/2` didn't preserve the order of its input variable list. The order of the elements in the returned list was not the same as in the input list.  
Own Id: OTP-3134  
Aux Id: seq1727
- The record message in `snmp_types.hrl` has been changed to reflect the new SNMPv3 message format. Specifically, the `community` field has changed name to `vs_n_hdr`. If the message version is v1 or v2c, it is still just the community string, but if the version is v3, it is a `v3_hdr` record. This change only affects applications that encodes / decodes SNMP messages using the module `snmp_pdu`s. If an application wants to use this new record (i.e. it needs to use SNMPv3), it must define the constant `SNMP_USE_V3` before the header file is included. This ensures that old applications that uses the old definition of `message` and don't use v3 don't have to be modified.

### SNMP Development Toolkit v3.0

#### Improvements and new features

- The agent is multi-lingual, and understands SNMPv1, SNMPv2c, and SNMPv3. Full SNMPv3 support is given, including encryption and authentication, but the optional proxy and notification filtering features are not implemented.
- A new function `snmp:send_notification/6` is added, which can be used to specify from which context a notification is sent.
- All relevant standard MIBs are included in the `mibs/` directory in the distribution, for reference purposes.
- The agent uses the application `crypto` for authentication and encryption. If these functions are to be used, the `crypto` application must be started before the `snmp` application.
- The manager supports SNMPv3. A few new options has been added to the start function `snmp_mgr:start/1`.

## Reported Fixed Bugs and Malfunctions

- The Mib Compiler now handles forward references in OBJECT IDENTIFIERS.  
Own Id: OTP-2942  
Aux Id: seq1415
- GET-NEXT on an empty table implemented with `snmp_generic` returns correctly.  
Own Id: OTP-2979  
Aux Id: seq1496
- The functions `snmp:log_to_text/2,3` can be used when the log is opened and in use by the agent.  
Own Id: OTP-2994  
Aux Id: seq1508

## Incompatibilities with v2.2.x

- The OTP-SNMPEA-MIB is not used anymore. The functionality is provided by standard MIBs. This means that managers that are using this MIB should use the standard MIBs instead. Applications that include the header file `OTP-SNMPEA-MIB.hrl` need to be changed.
- The handling of data in the configuration files vs. data in the actual tables is reworked and made consistent. This means that data changed by a manager survives a reboot, if the manager so wishes (by using the `StorageType` columns in the tables). There is a new configuration parameter, `force_config_load` which can be set to `true` to get the old behaviour, i.e. that the configuration files are always read at startup.
- The configuration files `address.conf`, `view.conf` and `trap_dest.conf` are not used anymore. The formats of `agent.conf` and `community.conf` have changed. `context.conf`, `usm.conf` and `vacm.conf` are new files. If the agent is started with the old files, it will convert them to the new files. Note however, that all information is not available in the old files. Thus, it is recommended to check the configuration files manually, or re-generate them using `snmp:config/0`.
- The module `snmp_mpc` has changed name to `snmp_mpd`, and it is now documented. It should be called only from a `net if` process.

## SNMP Development Toolkit v2.2.3

### Reported Fixed Bugs and Malfunctions

- When issuing a GET-NEXT request which contains more than one column from a table, and the table is empty, an erroneous GET-RESPONSE was sent to the manager. The agent failed to locate the next object for all columns.  
Own Id: OTP-2979

## SNMP Development Toolkit v2.2.2

### Reported Fixed Bugs and Malfunctions

- The function `snmp:validate_date_and_time/1` now works for all valid times.  
Own Id: OTP-2776  
Aux Id: seq1238



- The MIB compiler now handles DEFVALs specified as bit strings (e.g. '10010'b.  
Own Id: OTP-2826  
Aux Id: seq1285
- The MIB compiler now handles DEFVALs for opaque objects. Such a DEFVAL must be in string format, e.g. 'c0fa'H,  
Own Id: OTP-2827  
Aux Id: seq1285
- The MIB compiler now handles DEFVALs with no bits set for BITS objects.  
Own Id: OTP-2827  
Aux Id: seq1285
- The MIB compiler gives better error messages when erroneous DEFVALs are detected.  
Own Id: OTP-2827  
Aux Id: seq1285
- The MIB compiler gives better error messages when erroneous table definitions are detected.  
Own Id: OTP-2832

## SNMP Development Toolkit v2.2.1

### Reported Fixed Bugs and Malfunctions

- The example mib EX1-MIB in the user's guide compiles.  
Own Id: OTP-2576  
Aux Id: seq933
- The supervisor of the mibs has a correct shutdown time (infinity instead of 2 secs).  
Own Id: OTP-2606  
Aux Id: seq947
- The automatic conversion of trap\_dest.conf to the new notify.conf, target\_addres.conf, and target\_params.conf now gives an error message if the conversion fails.  
Own Id: OTP-2647
- The UDP port used by the agent is now opened with the flag reuseaddr set to true.  
Own Id: OTP-2655
- authenticationFailure are now sent correctly.  
Own Id: OTP-2690
- A get operation that refers to a not-accessible object now returns the correct error code.  
Own Id: OTP-2691
- The counter snmpInTotalsetVars is now updated properly.  
Own Id: OTP-2694

## SNMP Development Toolkit v2.2

### Improvements and new features

- The agent and snmp\_mgr support Inform-Requests when sending SNMPv2 notifications, as defined in rfc2273. This means that it is now possible to send a notification and have the agent wait for an

acknowledgement from the manager. See the User's guide, "Notification Sending" for a description of the mechanism.

- The agent uses the SNMP-NOTIFICATION-MIB and SNMP-TARGET-MIB from rfc2273 to select notification destinations, instead of the `intTrapDestTable` from OTP-SNMPEA-MIB.
- There are a few new functions in the module `snmp` for trap/notification sending.
- Two new modules are added; `snmp_notification_mib` and `snmp_target_mib`. These modules contain functions for initialising the agent data from configuration files.
- There is a new function `reconfigure/1` in `snmp_standard_mib`, which reconfigures the persistent objects from the configuration files.
- The following mibs are added to the distribution for completeness:
  - SNMPv2-TM
  - SNMPv2-SMI
  - SNMPv2-CONF
  - SNMP-FRAMEWORK-MIB
  - RFC1155-SMI
  - RFC-1212
  - RFC-1215

## Reported Fixed Bugs and Malfunctions

- The `snmp_mgr` no longer crashes if an unknown trap is received.  
Own Id: OTP-2137, OTP-2464
- The supervisor of the mibs has a correct shutdown time (infinity instead of 2 secs).  
Own Id: OTP-2310  
Aux Id: seq767, HA79462
- The Mib Compiler now handles the construct `MODULE OtherModule` in the `MODULE-COMPLIANCE` macro.  
Own Id: OTP-2465
- The Mib Compiler now handles the construct `INTEGER (0..1 | 3..4)`.  
Own Id: OTP-2466
- The Mib Compiler now handles `IMPLIED` OIDs  
Own Id: OTP-2467
- The Mib Compiler now handles `REFERENCE` in the `OBJECT-IDENTITY` macro.  
Own Id: OTP-2468

## Incompatibilities with v2.1.1

- The table `intTrapDestTable` in OTP-SNMPEA-MIB is not used anymore.
- There are three new configuration files; `target_addr.conf`, `target_params.conf` and `notify.conf`. The old file `trap_dest.conf` is not used. However, if a `trap_dest.conf` file exists, the agent automatically converts that file to the new files.
- The function `snmp:send_trap` no longer takes a `CommunityString` as a parameter. This parameter is now optional, and refers to the `snmpNotifyName` in `snmpNotifyTable`. Old code that uses the old function does not have to be changed.
- There are two new messages a `net_if` process must handle. These are messages to handle `Inform-Requests`.

## Future Improvements

More MIBs from the so-called SNMPv3 effort will be implemented, as these mibs become standards (although these MIBs don't require SNMPv3). This means that more tables from OTP-SNMPEA-MIB become obsolete, e.g. `intCommunityTable`, `intAddressTable` and `intViewTable`.

SNMPv3 itself will be implemented.

## SNMP Development Toolkit v2.1.1

### Reported Fixed Bugs and Malfunctions

- The `snmp_note_store` server sometimes didn't invoke its gc, which meant that its internal data structure could grow very large.  
Own Id: OTP-1946, OTP-2004  
Aux Id: seq451
- If a (possibly user-defined) `net_if` process crashes and restarts, the agent could crash as well.  
Own Id: OTP-2017
- Failover/takeover of the SNMP agent administering the MIB did not work, since the old implementation tried to delete a Mnesia table on a node which was not up and running (which always is the case at failover). Now the SNMP related Mnesia table is defined to have the `local_content` property.  
Own Id: OTP-2146  
Aux Id: seq1

## SNMP Development Toolkit v2.1.1

### Reported Fixed Bugs and Malfunctions

- The `snmp_note_store` server sometimes didn't invoke its gc, which meant that its internal data structure could grow very large.  
Own Id: OTP-1946, OTP-2004  
Aux Id: seq451
- If a (possibly user-defined) `net_if` process crashes and restarts, the agent could crash as well.  
Own Id: OTP-2017
- Failover/takeover of the SNMP agent administering the MIB did not work, since the old implementation tried to delete a Mnesia table on a node which was not up and running (which always is the case at failover). Now the SNMP related Mnesia table is defined to have the `local_content` property.  
Own Id: OTP-2146  
Aux Id: seq1

## SNMP Development Toolkit v2.1

### Improvements and new features

- The agent does not use the application sockets anymore. Thus, sockets does not have to be started before the agent (or manager).
- There are several new functions in the module `snmp` to check the syntax of and convert to/from `DateAndTime` structures.

### Reported Fixed Bugs and Malfunctions

- If an `is_set_ok` or `undo` instrumentation function crashed because it called an undefined function, the agent treated this as if the instrumentation function succeeded.  
Own Id: OTP-1762  
Aux Id: seq369
- The function `snmp:date_and_time/0` returns a correct `DateAndTime` structure. Previously, the sign in position 9 was wrong.  
Own Id: OTP-1838
- Traps larger than the maximum packet size are now handled.  
Own Id: OTP-1850  
Aux Id: seq416
- The `snmp_note_store` server never invoked its `gc`, which meant that its internal data structure could grow very large.  
Own Id: OTP-1946  
Aux Id: seq451
- Some objects in traps in SNMP v1 MIBs did not get the correct type in the compiled mib. This could lead to that the trap could not be sent.  
Own Id: OTP-2063

## SNMP Development Toolkit v2.0

### Improvements and new features

- The agent is bilingual, and understands SNMPv1 and SNMPv2c.
- The agent is made multi-threaded (configurable option).
- The agent can be configured to log requests to an Audit Trail Log.
- The application is restructured to be able to restart large parts of the agent in case of an error.
- The `NetIf` process API is simpler than in previous versions.
- New functions for handling AGENT-CAPABILITIES statements in the module `snmp`.
- A few new MIBs are added to the distribution. These are SNMPv2-MIB and SNMPv2-TC. The INTERNAL-MIB has changed its name to OTP-SNMPEA-MIB.
- Added the `coldStart` and `warmStart` traps to the STANDARD-MIB. None of them are sent by default though. The STANDARD-MIB and SNMPv2-MIB defines the same objects and notifications, except for the object specific to v1 and v2 respectively.
- `snmp:enum_to_int/2` and `snmp:int_to_enum/2` works for types as well as for objects.
- `RowStatus` no longer needs to be the last column for `snmp_generic` to work. This allows a table to be extended at a later stage.

- A table doesn't need to have all columns defined. Example: `snmp_generic` can handle a table with columns 1,2,3,5.
- `snmp_generic` works even though some columns in the middle of a table are not-accessible.
- Default values for strings may use "" syntax.
- The Mib Compiler understands both SNMPv1 and SNMPv2 MIBs and implements cross-version-IMPORTs.
- The Mib Compiler produces several error messages at the same time.
- The Mib Compiler is more strict.
- `snmp_mgr` can send and receive SNMPv2 messages.
- `snmp:config/0` generates a `sys.config` file.
- Added a record definition for the Mnesia table `snmp_variables` to `snmp_types.hrl`.
- The master agent may now be a distributed Erlang applications, and it reloads loaded mib during takeover.

## Reported Fixed Bugs and Malfunctions

- The Mib Compiler and `snmp_generic` handle objects with STATUS deprecated and obsolete.  
Own Id: OTP-1372  
Aux Id: HA47707
- Agent restarts after crashes keeps information about loaded mibs. Previously, the agent restarted, but all loaded mibs were unloaded.  
Own Id: OTP-1423  
Aux Id: AD84262
- The generated `.hrl` files now starts with an `-ifndef` which makes it possible to include the same file several times.  
Own Id: OTP-1425
- The agent could under som circumstances crash when a table instrumentation function crashed.  
Own Id: OTP-1738  
Aux Id: seq351

## Incompatibilities with v1.3.1

- The MIBs must be recompiled.
- The MIB compiler is more strict. Now it also checks that macros are imported correctly.
- The NetIf API is entirely re-written. It is now easier to implement a NetIf process.
- The Proxy handling in `snmp_net_if` is removed.
- The INTERNAL-MIB has changed its name to OTP-SNMPEA-MIB.

## Known bugs and problems

The User's Guide now describes where the Agent and MIB compiler don't implement the standard.

## SNMP Development Toolkit v1.3.1

### Reported Fixed Bugs and Malfunctions

- When trying to create a row that already exists in `snmp_local.db`, the new row is inserted instead of the old row.

## SNMP Development Toolkit v1.3

### Improvements and new features

- The MIB compiler has a new “include\_lib”-like option called `il`.  
Own Id: OTP-1041
- The RFC1213-MIB, STANDARD-MIB and INTERNAL-MIB are delivered in compiled form (`.bin`) as well as source form. The compiled mibs are located in `snmp-1.2.3/priv/mibs` directory. Use the option `il` to the mib compiler to find these files.  
Own Id: OTP-1272
- A new module `snmp_index` for building snmp indexes like the one used by Mnesia is added.  
Own Id: OTP-1307
- `snmp:mib_to_hrl` generates OBJECT IDENTIFIERS for all internal nodes in a MIB module, not just for the leaves in the tree.  
Own Id: OTP-1319

### Reported Fixed Bugs and Malfunctions

- The agent now treats the SNMPv2 error value `wrongValue` correctly.  
Own Id: OTP-1162
- It is now possible to load MIBs with short names.  
Own Id: OTP-1195
- The MIB-compiler now compiles full-path files.  
Own Id: OTP-1217
- The function `snmp:config/0` works for WinNT.  
Own Id: OTP-1216
- The agent handles bad index returned from table instrumentation functions.  
Own Id: OTP-1222
- The data type `IpAddress` works better.  
Own Id: OTP-1234
- The function `snmp:date_and_time/0` now works correctly when called at midnight.  
Own Id: OTP-1236
- The documentation of `snmp_generic:table_set_elements/3` is updated to reflect the fact that it is the caller's responsibility to call this function from within a transaction, if Mnesia is used.  
Own Id: OTP-1265
- The MIB compiler prints the error message “Unexpected 0” when an INTEGER begins with 0.  
Own Id: OTP-1274
- The agent now decodes negative integers correctly.  
Own Id: OTP-1298

- Trying to delete a non-existing row implemented with `snmp_generic` now returns `noError` instead of `inconsistentValue`.  
Own Id: OTP-1331
- Default value initialisation now works for Mnesia tables when `snmp_generic:table_set_row/5` is called.  
Own Id: OTP-1338
- Trying to set a row implemented with `snmp_generic` when an INDEX column was read-write got the agent into an infinite loop.  
Own Id: OTP-1342
- `mib_to_hrl` doesn't generate enum definitions for imported OBJECT-TYPES.  
Own Id: OTP-1355
- Traps are not only sent to one manager if several managers are specified as trap destinations, but to all of them.  
Own Id: OTP-1366

## SNMP Development Toolkit v1.2.1

### Improvements and new features

- The file `STANDARD-MIB.hrl` is now included in the directory `snmp/include`. It may be included in source code with `-include_lib("snmp/include/STANDARD-MIB.hrl")`.  
Own Id: OTP-1063  
Aux Id: HA34986, HA36413

### Reported Fixed Bugs and Malfunctions

- Calls to the agent had too short timeout. This was most obvious in calls to `snmp:load_mibs/2`.  
Own Id: OTP-1078, OTP-1096  
Aux Id: HA36413
- `snmp:config/0` tried to validate correct IP addresses, resulting in an error when DNS was not used.  
Own Id: OTP-1094
- `snmp:mib_to_hrl/1` generated entries for imported OBJECT IDENTIFIERS and imported DEFVALs, which could result in macro conflict when `.hrl` files from more than one MIB was included.  
Own Id: OTP-1126
- When a row was `createAndWait`d, it wasn't possible to set a row to active in the same request as the rest of the mandatory columns were set.  
Own Id: OTP-1128
- `snmp:int_to_enum/2` crashed when called with an Oid without any enums.  
Own Id: OTP-1129
- The agent could crash if a RowIndex in a set-request was of bad type.  
Own Id: OTP-1131

## SNMP Development Toolkit v1.2

### Improvements and new features

- A new set-phase `undo` is added. When the agent has called `is_set_ok` for an object, it will always call either `set` or `undo` at a later stage. This may be used to reserve resources in the `is_set_ok` function.
- More documentation on Mnesia.
- Added conversion functions from OBJECT IDENTIFIER to symbolic name, and vice versa. These are called `snmp:oid_to_name/1` and `snmp:name_to_oid/1`.
- Added conversion functions from enumerated integers to symbolic value, and vice versa. These are called `snmp:int_to_enum/2` and `snmp:enum_to_int/2`.
- Added an extra parameter from the (configurable) Net If process for each SNMP packet. This value can be retrieved by `snmp:current_net_if_data/0` from an instrumentation function.
- Added a function `snmp:date_and_time/0`, which returns a `DateAndTime` value, according to RFC1903.
- A new function `snmp:str_apply/1` may be used to pass options to the Mib Compiler from the unix command line.
- There is now no need to have a `RowStatus` column for a read-only table that is implemented with `snmp_generic`.
- A Mnesia table that implements an SNMP table with `snmp_generic` may have more columns than the SNMP table.
- Added a function `snmp:get/2` to retrieve values from the agent from programs.
- The `snmp:config/0` tool handles host names as strings better.
- The Mib Compiler generates better warnings.
- The agent handles atoms as return values from enumerated integers better.

### Reported Fixed Bugs and Malfunctions

- DEFVALs for strings are handled correctly by the Mib Compiler.
- The SNMPv2 error code `notWritable` is translated to the SNMPv1 code `noSuchName`.
- The instrumentation function `delete` is now called correctly when unloading a Mib.
- An `is_set_ok` function is added for `snmp_generic:variable_func`.
- `snmp:mib_to_hrl` doesn't generate output for imported MIB objects.
- The SNMP type `IpAddress` is now BER encoded correctly.
- The agent could hang when last object in the MIB was `not-accessible`, and a GET-NEXT request was received.
- The Mib Compiler doesn't crash when the MIB file doesn't exist.
- The Mib Compiler doesn't crash when an instrumentation function is specified for a column object in a table.

### Incompatibilities with v1.1

- The MIBs must be recompiled.



- 
- The contents of the output file from `snmp:mib_to_hrl` has changed. Imported objects are not included. All MIBs with imported objects must be run through `snmp:mib_to_hrl`, and included in the Erlang code as well.
  - The start functions for the agent has changed. Note that this is invisible when Erlang 4.4 is used.

# SNMP Reference Manual

## Short Summaries

- Application **snmp** [page 92] – The SNMP Application
- Erlang Module **snmp** [page 94] – Interface Functions to the SNMP toolkit
- Erlang Module **snmp\_community\_mib** [page 104] – Instrumentation Functions for SNMP-COMMUNITY-MIB
- Erlang Module **snmp\_error** [page 105] – Functions for Reporting SNMP Errors
- Erlang Module **snmp\_framework\_mib** [page 106] – Instrumentation Functions for SNMP-FRAMEWORK-MIB
- Erlang Module **snmp\_generic** [page 107] – Generic Functions for Implementing SNMP Objects in a Database
- Erlang Module **snmp\_index** [page 111] – Abstract Data Type for SNMP Indexing
- Erlang Module **snmp\_local\_db** [page 115] – The SNMP built-in database
- Erlang Module **snmp\_mgr** [page 118] – SNMP Manager
- Erlang Module **snmp\_mpd** [page 122] – Message Processing and Dispatch module for SNMP
- Erlang Module **snmp\_notification\_mib** [page 124] – Instrumentation Functions for SNMP-NOTIFICATION-MIB
- Erlang Module **snmp\_pdus** [page 125] – Encode and Decode Functions for SNMP PDUs
- Erlang Module **snmp\_standard\_mib** [page 128] – Instrumentation Functions for STANDARD-MIB and SNMPv2-MIB
- Erlang Module **snmp\_supervisor** [page 130] – A supervisor for the SNMP Processes
- Erlang Module **snmp\_target\_mib** [page 132] – Instrumentation Functions for SNMP-TARGET-MIB
- Erlang Module **snmp\_user\_based\_sm\_mib** [page 133] – Instrumentation Functions for SNMP-USER-BASED-SM-MIB
- Erlang Module **snmp\_view\_based\_acm\_mib** [page 134] – Instrumentation Functions for SNMP-VIEW-BASED-ACM-MIB

## **snmp**

No functions are exported

## snmp

The following functions are exported:

- `add_agent_caps(SysORID, SysORDescr) -> SysORIndex`  
[page 94] Adds an AGENT-CAPABILITY definition to the agent
- `c(File)`  
[page 94] Compiles the specified MIB
- `c(File,Options) -> {ok, BinFileName} | {error, Reason}`  
[page 94] Compiles the specified MIB
- `change_log_size(NewSize) -> ok | {error, Reason}`  
[page 95] Changes the size of the Audit Trail Log
- `config() -> ok | {error, Reason}`  
[page 95] A simple SNMP agent configuration tool
- `current_address() -> {value, {IP, UDP}} | false`  
[page 95] The ip address of the manager
- `current_community() -> {value, Community} | false`  
[page 96] The community of the current request
- `current_context() -> {value, ContextName} | false`  
[page 96] The context of the current request
- `current_net_if_data() -> {value, NetIfData} | false`  
[page 96] The Net\_if data of the current pdu
- `current_request_id() -> {value, RequestId} | false`  
[page 96] The request Id of the current request
- `date_and_time() -> DateAndTime`  
[page 96] Current date and time as an OCTET STRING
- `date_and_time_to_universal_time(DateAndTime) -> UTC`  
[page 97] Converts a DateAndTime value to UTC
- `debug(Agent,Bool) -> void()`  
[page 97] Turns debugging on/off
- `del_agent_caps(SysORIndex) -> void()`  
[page 97] Deletes an AGENT-CAPABILITY definition from the agent
- `enum_to_int(Name,Enum) -> {value, Int} | false`  
[page 97] Converts an enum value to an integer
- `get(Agent,Vars) -> Values | {error, Reason}`  
[page 97] Performs a get operation on the agent
- `get_agent_caps() -> [[SysORIndex, SysORID, SysORDescr, SysORUpTime]]`  
[page 97] Returns all AGENT-CAPABILITY definitions in the agent
- `info(Agent) -> [{Key, Value}]`  
[page 98] Returns information about the agent
- `int_to_enum(Name,Int) -> {value, Enum} | false`  
[page 98] Converts an integer to an enum value
- `is_consistent(Mibs) -> ok | {error, Reason}`  
[page 98] Checks for OID conflicts between MIBs
- `load_mibs(Agent,Mibs) -> ok | {error, Reason}`  
[page 98] Loads MIBs into the agent

- `local_time_to_date_and_time(Local) -> DateAndTime`  
[page 98] Converts a Local time value to DateAndTime
- `log_to_txt(LogDir, Mibs)`  
[page 99] Converts an Audit Trail Log to text format
- `log_to_txt(LogDir, Mibs, OutFile) -> ok | {error, Reason}`  
[page 99] Converts an Audit Trail Log to text format
- `mib_to_hrl(MibName) -> ok | {error, Reason}`  
[page 99] Generates constants for the objects in the MIB
- `name_to_oid(Name) -> {value, oid()} | false`  
[page 99] Converts a symbolic name to an OID
- `oid_to_name(OID) -> {value, Name} | false`  
[page 99] Converts an OID to a symbolic name
- `register_subagent(Agent, SubTreeOid, Subagent) -> ok | {error, Reason}`  
[page 99] Registers a subagent under a subtree
- `send_notification(Agent, Notification, Receiver)`  
[page 100] Sends a notification
- `send_notification(Agent, Notification, Receiver, Varbinds)`  
[page 100] Sends a notification
- `send_notification(Agent, Notification, Receiver, NotifyName, Varbinds)`  
[page 100] Sends a notification
- `send_notification(Agent, Notification, Receiver, NotifyName, ContextName, Varbinds) -> void()`  
[page 100] Sends a notification
- `send_trap(Agent, Trap, Community)`  
[page 101] Sends a trap
- `send_trap(Agent, Trap, Community, Varbinds) -> void()`  
[page 101] Sends a trap
- `universal_time_to_date_and_time(UTC) -> DateAndTime`  
[page 102] Converts a UTC value to DateAndTime
- `unload_mibs(Agent, Mibs) -> ok | {error, Reason}`  
[page 102] Unloads MIBs from the agent
- `unregister_subagent(Agent, SubagentOidOrPid) -> ok | {ok, SubAgentPid} | {error, Reason}`  
[page 103] Unregisters a subagent
- `validate_date_and_time(DateAndTime) bool()`  
[page 103] Checks if a DateAndTime value is correct

## **snmp\_community\_mib**

The following functions are exported:

- `configure(ConfDir) -> void()`  
[page 104] Configures the SNMP-COMMUNITY-MIB
- `reconfigure(ConfDir) -> void()`  
[page 104] Configures the SNMP-COMMUNITY-MIB

## snmp\_error

The following functions are exported:

- `config_err(Format,Args) -> void()`  
[page 105] Called if a configuration error occurs
- `user_err(Format,Args) -> void()`  
[page 105] Called if a user related error occurs

## snmp\_framework\_mib

The following functions are exported:

- `configure(ConfDir) -> void()`  
[page 106] Configures the SNMP-FRAMEWORK-MIB
- `init() -> void()`  
[page 106] Initializes the SNMP-FRAMEWORK-MIB

## snmp\_generic

The following functions are exported:

- `get_status_col(Name,Cols)`  
[page 108] Gets the value of the status column from Cols
- `get_status_col(NameDb,Cols) -> {ok, StatusVal} | false`  
[page 108] Gets the value of the status column from Cols
- `table_func(Op1,NameDb)`  
[page 108] Default instrumentation function for tables
- `table_func(Op2,RowIndex,Cols,NameDb) -> Ret`  
[page 108] Default instrumentation function for tables
- `table_get_elements(NameDb,RowIndex,Cols) -> Values`  
[page 109] Gets elements in a table row
- `table_next(NameDb,RestOid) -> RowIndex | endOfTable`  
[page 109] Finds the next row in the table
- `table_row_exists(NameDb,RowIndex) -> bool()`  
[page 109] Checks if a row in a table exists
- `table_set_elements(NameDb,RowIndex,Cols) -> bool()`  
[page 109] Sets elements in a table row
- `variable_func(Op1,NameDb)`  
[page 109] Default instrumentation function for tables
- `variable_func(Op2,Val,NameDb) -> Ret`  
[page 109] Default instrumentation function for tables
- `variable_get(NameDb) -> {value, Value} | undefined`  
[page 109] Gets the value of a variable
- `variable_set(NameDb,NewVal) -> true | false`  
[page 109] Sets a value for a variable

## snmp\_index

The following functions are exported:

- `delete(Index) -> true`  
[page 112] Deletes an index table
- `delete(Index, Key) -> NewIndex`  
[page 113] Deletes an item from the index
- `get(Index, KeyOid) -> {ok, {KeyOid, Value}} | undefined`  
[page 113] Gets the item with KeyOid
- `get_last(Index) -> {ok, {KeyOid, Value}} | undefined`  
[page 113] Gets the last item in the index structure
- `get_next(Index, KeyOid) -> {ok, {NextKeyOid, Value}} | undefined`  
[page 113] Gets the next item
- `insert(Index, Key, Value) -> NewIndex`  
[page 113] Inserts an item into the index
- `key_to_oid(Index, Key) -> KeyOid`  
[page 113] Converts a key to an OBJECT IDENTIFIER
- `new(KeyTypes)`  
[page 114] Creates a new snmp index structure

## snmp\_local\_db

The following functions are exported:

- `dump() -> ok | {error, Reason}`  
[page 116] Dumps the database to disk
- `match(NameDb, Pattern)`  
[page 116] Performs an ets match on the table
- `print()`  
[page 116] Prints the database to screen
- `print(TableName)`  
[page 116] Prints the database to screen
- `print(TableName, Db)`  
[page 116] Prints the database to screen
- `table_create(NameDb) -> bool()`  
[page 116] Creates a table
- `table_create_row(NameDb,RowIndex,Row) -> bool()`  
[page 116] Creates a row in a table
- `table_delete(NameDb) -> void()`  
[page 116] Deletes a table
- `table_delete_row(NameDb,RowIndex) -> bool()`  
[page 116] Deletes the row in the table
- `table_exists(NameDb) -> bool()`  
[page 116] Checks if a table exists
- `table_get_row(NameDb,RowIndex) -> Row | undefined`  
[page 116] Gets a row from the table

## snmp\_mgr

The following functions are exported:

- `expect(Id, What) -> ok | {error, Id, Reason}`  
[page 119] Tests if the manager has received a response, trap, inform or report
- `expect(Id, ErrorStatus, ErrorIndex, Varbinds)`  
[page 119] Tests if the manager has received a response, trap, inform or report
- `expect(Id, trap, Enterp, Generic, Specific, Varbinds)`  
[page 119] Tests if the manager has received a response, trap, inform or report
- `expect(Id, v2trap, Varbinds)`  
[page 119] Tests if the manager has received a response, trap, inform or report
- `expect(Id, report, Varbinds)`  
[page 119] Tests if the manager has received a response, trap, inform or report
- `expect(Id, {inform, InformReply}, Varbinds)`  
[page 119] Tests if the manager has received a response, trap, inform or report
- `g(Oids) -> void()`  
[page 119] Sends a get-request
- `gb(NonRepeaters, MaxRepetitions, Oids) -> void()`  
[page 119] Sends a get-bulk-request
- `gn(Oids) -> void()`  
[page 120] Sends a get-next-request
- `gn() -> void()`  
[page 120] Sends a get-next-request
- `gn(N) -> void()`  
[page 120] Sends N get-next-request requests
- `r() -> void()`  
[page 120] Resends the last request
- `s(Varbinds) -> void()`  
[page 120] Sends a set-request
- `start(Options)`  
[page 120] Starts the SNMP manager
- `start_link(Options) -> void()`  
[page 120] Starts the SNMP manager
- `stop() -> void()`  
[page 121] Stops the SNMP manager

## snmp\_mpd

The following functions are exported:

- `init_mpd(Options) -> mpd_state()`  
[page 122] Initializes the MPD module
- `process_packet(Packet, TDomain, TAddress, State) -> {ok, Vsn, Pdu, PduMS, ACMDData} | {discarded, Reason}`  
[page 122] Processes a packet received from the network

- `generate_response_msg(Vsn, RePdu, Type, ACMDData) -> {ok, Packet} | {discarded, Reason}`  
[page 122] Generates a response packet to be sent to the network
- `generate_msg(Vsn, Pdu, MsgData, To) -> {ok, PacketsAndAddresses} | {discarded, Reason}`  
[page 123] Generates a request message to be sent to the network
- `discarded_pdu(Variable) -> void()`  
[page 123]

## **snmp\_notification\_mib**

The following functions are exported:

- `configure(ConfDir) -> void()`  
[page 124] Configures the SNMP-NOTIFICATION-MIB
- `reconfigure(ConfDir) -> void()`  
[page 124] Configures the SNMP-NOTIFICATION-MIB

## **snmp\_pdus**

The following functions are exported:

- `dec_message([byte()]) -> Message`  
[page 125] Decodes an SNMP Message
- `dec_message_only([byte()]) -> Message`  
[page 125] Decodes an SNMP Message, but not the data part
- `dec_pdu([byte()]) -> Pdu`  
[page 125] Decodes an SNMP Pdu
- `dec_scoped_pdu([byte()]) -> ScopedPdu`  
[page 126] Decodes an SNMP ScopedPdu
- `dec_scoped_pdu_data([byte()]) -> ScopedPduData`  
[page 126] Decodes an SNMP ScopedPduData
- `dec_usm_security_parameters([byte()]) -> UsmSecParams`  
[page 126] Decodes SNMP UsmSecurityParameters
- `enc_encrypted_scoped_pdu(EncryptedScopedPdu) -> [byte()]`  
[page 126] Encodes an encrypted SNMP scopedPDU
- `enc_message(Message) -> [byte()]`  
[page 126] Encodes an SNMP Message
- `enc_message_only(Message) -> [byte()]`  
[page 126] Encodes an SNMP Message, but not the data part
- `enc_pdu(Pdu) -> [byte()]`  
[page 126] Encodes an SNMP Pdu
- `enc_scoped_pdu(ScopedPdu) -> [byte()]`  
[page 126] Encodes an SNMP scopedPDU
- `enc_usm_security_parameters(UsmSecParams) -> [byte()]`  
[page 127] Encodes SNMP UsmSecurityParameters



## snmp\_standard\_mib

The following functions are exported:

- `configure(ConfDir) -> void()`  
[page 128] Configures the STANDARD-MIB and SNMPv2-MIB
- `inc(Name) -> void()`  
[page 128] Increments a variable in the MIB
- `inc(Name, N) -> void()`  
[page 128] Increments a variable in the MIB
- `reconfigure(ConfDir) -> void()`  
[page 128] Configures the STANDARD-MIB and SNMPv2-MIB
- `reinit() -> void()`  
[page 129] Resets all snmp counters to 0
- `sys_up_time() -> Time`  
[page 129] Gets the system up time

## snmp\_supervisor

The following functions are exported:

- `start_sub()`  
[page 130] Starts the SNMP supervisor for subagents only
- `start_sub(Opts) -> {ok, pid()} | {error, {already_started, pid()}} | {error, Reason}`  
[page 130] Starts the SNMP supervisor for subagents only
- `start_master(DbDir,ConfDir)`  
[page 130] Starts the SNMP supervisor for all agents
- `start_master(DbDir,ConfDir,Opts) -> {ok, pid()} | {error, {already_started, pid()}} | {error, Reason}`  
[page 130] Starts the SNMP supervisor for all agents
- `start_subagent(ParentAgent,Subtree,Mibs) -> {ok, pid()} | {error, Reason}`  
[page 131] Starts a subagent
- `stop_subagent(SubAgent) -> ok | no_such_child`  
[page 131] Stops a subagent

## snmp\_target\_mib

The following functions are exported:

- `configure(ConfDir) -> void()`  
[page 132] Configures the SNMP-TARGET-MIB
- `reconfigure(ConfDir) -> void()`  
[page 132] Configures the SNMP-TARGET-MIB

## **snmp\_user\_based\_sm\_mib**

The following functions are exported:

- `configure(ConfDir) -> void()`  
[page 133] Configures the SNMP-USER-BASED-SM-MIB
- `reconfigure(ConfDir) -> void()`  
[page 133] Configures the SNMP-USER-BASED-SM-MIB

## **snmp\_view\_based\_acm\_mib**

The following functions are exported:

- `configure(ConfDir) -> void()`  
[page 134] Configures the SNMP-VIEW-BASED-ACM-MIB
- `reconfigure(ConfDir) -> void()`  
[page 134] Configures the SNMP-VIEW-BASED-ACM-MIB

# 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. <! NOTE: This list is duplicated in `snmp_config.sgml` ->

`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.

`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`.

`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`.

## SEE ALSO

`application(3)`, `disk_log(3)`

# snmp (Module)

This module 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. This 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()}`
- `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-1.2.1/mibs/"]` (or what the current version may be in your 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 `group_check` specifies whether the mib compiler should check the `OBJECT-GROUP` macro for correctness or not. Default is `true`.

The MIB compiler understands both SMIV1 and SMIV2 MIBs. It uses the `MODULE-IDENTITY` statement to determine if the MIB is version 1 or 2.

The MIB compiler can be invoked from the OS command line by using the command `erlc`. `erlc` recognises the extension `.mib`, and invokes the SNMP MIB compiler for files with that extension. The options `db` and `group_check` 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)` for a description of how the log size is changed.

The change is permanent, as long as the log is not deleted. This means that the log size is remembered 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 which asks some questions and generates `sys.config` and `*.conf` files.

`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 is handling 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(DateAndTime) -> UTC`

Types:

- `DateAndTime` = `[int()]`
- `UTC` = `{{Y,Mo,D},{H,M,S}}`

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

`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.

`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.

`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.

`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.

`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(Local) -> DateAndTime`

Types:

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

- `DateAndTime = [int()]`

Converts a local time value to a `DateAndTime` list. 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}`

Types:

- `LogDir = string()`
- `Mibs = [MibName]`
- `OutFile = string()`
- `MibName = string()`

Converts an Audit Trail Log to a readable text file. The function can be used on a running system, or by copying the entire log directory and calling this function off-line.

`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"`.

`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 that the OBJECT IDENTIFIER given is for the object, not for an instance.

`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.

`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 first 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 in 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, snmp\_error:user\_err/2 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 universal 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.

## SEE ALSO

calendar(3), erlc(1)

# snmp\_community\_mib (Module)

This module 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 `snmp_error:config_err/2`, 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 the data in 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 `snmp_error:config_err/2`, 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`.

# snmp\_error (Module)

This module contains two callback functions which are called if an error occurs at different times during agent operation.

A simple implementation of this module is provided with the toolkit. In this implementation, the errors are sent to the `error_logger`. This module can be modified to customize the way errors are reported.

## 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\_framework\_mib (Module)

This module 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 the data in 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 `snmp_error:config_err/2`, 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()`

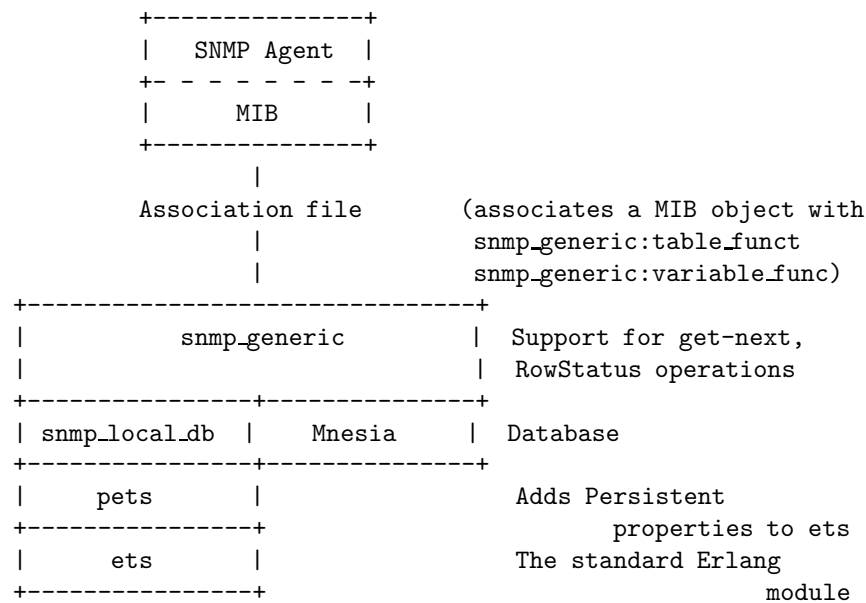
This function is called from the supervisor at system start-up.

Creates the necessary objects in the database if they do not exist. It does not destroy any old values.

## snmp\_generic (Module)

This module 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.

```
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 (Module)

This module 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 IDENTIFIER, 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 112] 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 (Module)

This module 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 a module, `ets` (Persistent ets (Erlang Term Store)), that keeps an ets database in memory for speed, and on disk for persistent storage. At start-up, the ets database is initialized from disk. All readings from the database go to the ets. Writings go to the ets, and a note is made in a log file. When the database is closed, the entire ets database is dumped, and the log file is cleared. If a crash occurs, the log file will always contain the latest changes. At start-up, the dumped ets is read, and then the database is recovered using the log file. It is also possible to manually dump the database, in which case the entire ets is dumped, and the log file is cleared.

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.

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 the table. `Cols` is a list of column numbers in case of a get operation, and a list of column numbers and values in case of a set operation.

## Exports

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

This function can be used to dump the database at any time. The entire ets database is dumped to disk, and the log file is cleared. This might be useful if the log file grows rapidly. Returns `{error, Reason}` if a file system error occurred.

`match(NameDb, Pattern)`

Performs an ets matching on the table. See the man page of the Erlang 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-SNMP-EA-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.

## SEE ALSO

ets(3), snmp\_generic(3)

## snmp\_mgr (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 {error, Id, Reason} is returned. A call to expect is normally directly preceded 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()`

Resends the last request.

`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 isn't 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.

Mandatory 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 it is `pdu` (default), message is `{snmp_pdu, PDU}` where PDU is a `pdu()` or a `trappdu()` record defined in `snmp_types.hrl`. If it is `msg`, message is `{snmp_msg, Msg, Ip, Udp}`. Default is that 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`.

`stop() -> void()`

Stops the SNMP manager.



# snmp\_mpd (Module)

This module 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, ACMDData} | {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()
- ACMDData = 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, ACMDData) -> {ok, Packet} | {discarded, Reason}
```

Types:

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

- Type = atom()
- ACMDData = 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\_addr()]
- PacketsAndAddresses = [{TDomain, TAddress, Packet}]
- TDomain = snmpUDPDDomain
- 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 (Module)

This module 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 `snmp_error:config_err/2`, 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 the data in the configuration files.

If an error is found in the configuration file, it is reported using the function `snmp_error:config_err/2`, 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`.

# snmp\_pdus (Module)

RFC1157, RFC1905 and/or RFC2272 should be studied carefully before using this module.

This module 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 that if it 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 that 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. This 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 a SNMP UsmSecurityParameters

`enc_encrypted_scoped_pdu(EncryptedScopedPdu) -> [byte()]`

Types:

- `EncryptedScopedPdu = [byte()]`

Encodes an encrypted SNMP ScopedPdu into an OCTET STRING that can be used as the data field in a message record, that later can be encoded with a call to `enc_message_only/1`.

This function should be used whenever the ScopedPDU is encrypted.

`enc_message(Message) -> [byte()]`

Types:

- `Message = #message`

Encodes a message record to a list of bytes.

`enc_message_only(Message) -> [byte()]`

Types:

- `Message = #message`

`Message` is a record where the data field is assumed to be encoded (a list of bytes). If it's a v1 or v2 message, the data field is an encoded PDU. If it's a v3 message, data is an encoded and possibly encrypted scopedPDU.

`enc_pdu(Pdu) -> [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. This list of bytes 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 (Module)

This module 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 `snmp_error:config_err/2`, 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: `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 the data in 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 `snmp_error:config_err/2`, 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: `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 (Module)

This 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. This 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()]`
- `NetIfModule = atom()`
- `Name = {local, atom()} | {global, atom()}`

Starts a supervisor for the SNMP agent system. This 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 are 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 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 isn't running, the function fails with reason badarg.

# snmp\_target\_mib (Module)

This module 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 `snmp_error:config_err/2`, 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 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 in 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 `snmp_error:config_err/2`, 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 files read are: `target_addr.conf` and `target_params.conf`.

# snmp\_user\_based\_sm\_mib (Module)

This module 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 `snmp_error:config_err/2`, 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: `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 in 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 `snmp_error:config_err/2`, 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: `usm.conf`.

# snmp\_view\_based\_acm\_mib (Module)

This module 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 `snmp_error:config_err/2`, 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: `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 in 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 `snmp_error:config_err/2`, 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: `vacm.conf`.





# List of Figures

## Chapter 1: SNMP User's Guide

1.1	MIB Compiler Principles . . . . .	7
1.2	Starting the Agent . . . . .	7
1.3	Architecture . . . . .	8
1.4	Contents of my_table . . . . .	20
1.5	GetNext from [3,1,1] and [5,1,1]. . . . .	21
1.6	GetNext from [3,2,1] and [5,2,1]. . . . .	21
1.7	GetNext from [3,1,2] and [4,1,2]. . . . .	22
1.8	The Purpose of Net if . . . . .	58





# List of Tables

**Chapter 1: SNMP User's Guide**

1.1 - ..... 62



# Index

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

add\_agent\_caps/2  
    *snmp* , 94

c/1  
    *snmp* , 94

c/2  
    *snmp* , 94

change\_log\_size/1  
    *snmp* , 95

config/0  
    *snmp* , 95

config\_err/2  
    *snmp\_error* , 105

configure/1  
    *snmp\_community\_mib* , 104  
    *snmp\_framework\_mib* , 106  
    *snmp\_notification\_mib* , 124  
    *snmp\_standard\_mib* , 128  
    *snmp\_target\_mib* , 132  
    *snmp\_user\_based\_sm\_mib* , 133  
    *snmp\_view\_based\_acm\_mib* , 134

current\_address/0  
    *snmp* , 95

current\_community/0  
    *snmp* , 96

current\_context/0  
    *snmp* , 96

current\_net\_if\_data/0  
    *snmp* , 96

current\_request\_id/0  
    *snmp* , 96

date\_and\_time/0  
    *snmp* , 96

date\_and\_time\_to\_universal\_time/1  
    *snmp* , 97

debug/2  
    *snmp* , 97

dec\_message/1  
    *snmp\_pdus* , 125

dec\_message\_only/1  
    *snmp\_pdus* , 125

dec\_pdu/1  
    *snmp\_pdus* , 125

dec\_scoped\_pdu/1  
    *snmp\_pdus* , 126

dec\_scoped\_pdu\_data/1  
    *snmp\_pdus* , 126

dec\_usm\_security\_parameters/1  
    *snmp\_pdus* , 126

del\_agent\_caps/1  
    *snmp* , 97

delete/1  
    *snmp\_index* , 112

delete/2  
    *snmp\_index* , 113

discarded\_pdu/1  
    *snmp\_mpd* , 123

dump/0  
    *snmp\_local\_db* , 116

enc\_encrypted\_scoped\_pdu/1  
    *snmp\_pdus* , 126

enc\_message/1  
    *snmp\_pdus* , 126

enc\_message\_only/1  
    *snmp\_pdus* , 126

enc\_pdu/1  
    *snmp\_pdus* , 126

enc\_scoped\_pdu/1  
    *snmp\_pdus* , 126

enc\_usm\_security\_parameters/1  
    *snmp\_pdus* , 127

enum\_to\_int/2  
    *snmp* , 97

expect/2  
    *snmp\_mgr* , 119

expect/3  
    *snmp\_mgr* , 119

expect/4  
    *snmp\_mgr* , 119

expect/6  
    *snmp\_mgr* , 119

g/1  
    *snmp\_mgr* , 119

gb/3  
    *snmp\_mgr* , 119

generate\_msg/4  
    *snmp\_mpd* , 123

generate\_response\_msg/4  
    *snmp\_mpd* , 122

get/2  
    *snmp* , 97  
    *snmp\_index* , 113

get\_agent\_caps/0  
    *snmp* , 97

get\_last/1  
    *snmp\_index* , 113

get\_next/2  
    *snmp\_index* , 113

get\_status\_col/2  
    *snmp\_generic* , 108

gn/0  
    *snmp\_mgr* , 120

gn/1  
    *snmp\_mgr* , 120

inc/1  
    *snmp\_standard\_mib* , 128

inc/2  
    *snmp\_standard\_mib* , 128

info/1  
    *snmp* , 98

init/0  
    *snmp\_framework\_mib* , 106

init\_mpd/1  
    *snmp\_mpd* , 122

insert/3  
    *snmp\_index* , 113

int\_to\_enum/2  
    *snmp* , 98

is\_consistent/1  
    *snmp* , 98

key\_to\_oid/2  
    *snmp\_index* , 113

load\_mibs/2  
    *snmp* , 98

local\_time\_to\_date\_and\_time/1  
    *snmp* , 98

log\_to\_txt/2  
    *snmp* , 99

log\_to\_txt/3  
    *snmp* , 99

match/2  
    *snmp\_local\_db* , 116

mib\_to\_hrl/1  
    *snmp* , 99

name\_to\_oid/1  
    *snmp* , 99

new/1  
    *snmp\_index* , 114

oid\_to\_name/1  
    *snmp* , 99

print/0  
    *snmp\_local\_db* , 116

print/1  
    *snmp\_local\_db* , 116

print/2  
    *snmp\_local\_db* , 116

process\_packet/4  
    *snmp\_mpd* , 122

- 
- r/0
    - snmp\_mgr* , 120
  - reconfigure/1
    - snmp\_community\_mib* , 104
    - snmp\_notification\_mib* , 124
    - snmp\_standard\_mib* , 128
    - snmp\_target\_mib* , 132
    - snmp\_user\_based\_sm\_mib* , 133
    - snmp\_view\_based\_acm\_mib* , 134
  - register\_subagent/3
    - snmp* , 99
  - reinit/0
    - snmp\_standard\_mib* , 129
  - s/1
    - snmp\_mgr* , 120
  - send\_notification/3
    - snmp* , 100
  - send\_notification/4
    - snmp* , 100
  - send\_notification/5
    - snmp* , 100
  - send\_notification/6
    - snmp* , 100
  - send\_trap/3
    - snmp* , 101
  - send\_trap/4
    - snmp* , 101
  - snmp*
    - add\_agent\_caps/2, 94
    - c/1, 94
    - c/2, 94
    - change\_log\_size/1, 95
    - config/0, 95
    - current\_address/0, 95
    - current\_community/0, 96
    - current\_context/0, 96
    - current\_net\_if\_data/0, 96
    - current\_request\_id/0, 96
    - date\_and\_time/0, 96
    - date\_and\_time\_to\_universal\_time/1, 97
    - debug/2, 97
    - del\_agent\_caps/1, 97
    - enum\_to\_int/2, 97
    - get/2, 97
    - get\_agent\_caps/0, 97
    - info/1, 98
    - int\_to\_enum/2, 98
    - is\_consistent/1, 98
    - load\_mibs/2, 98
    - local\_time\_to\_date\_and\_time/1, 98
    - log\_to\_txt/2, 99
    - log\_to\_txt/3, 99
    - mib\_to\_hrl/1, 99
    - name\_to\_oid/1, 99
    - oid\_to\_name/1, 99
    - register\_subagent/3, 99
    - send\_notification/3, 100
    - send\_notification/4, 100
    - send\_notification/5, 100
    - send\_notification/6, 100
    - send\_trap/3, 101
    - send\_trap/4, 101
    - universal\_time\_to\_date\_and\_time/1, 102
    - unload\_mibs/2, 102
    - unregister\_subagent/2, 103
    - validate\_date\_and\_time/1, 103
  - snmp\_community\_mib*
    - configure/1, 104
    - reconfigure/1, 104
  - snmp\_error*
    - config\_err/2, 105
    - user\_err/2, 105
  - snmp\_framework\_mib*
    - configure/1, 106
    - init/0, 106
  - snmp\_generic*
    - get\_status\_col/2, 108
    - table\_func/2, 108
    - table\_func/4, 108
    - table\_get\_elements/3, 109
    - table\_next/2, 109
    - table\_row\_exists/2, 109
    - table\_set\_elements/3, 109
    - variable\_func/2, 109
    - variable\_func/3, 109
    - variable\_get/1, 109
    - variable\_set/2, 109
  - snmp\_index*
    - delete/1, 112
    - delete/2, 113
    - get/2, 113
    - get\_last/1, 113
    - get\_next/2, 113
    - insert/3, 113
    - key\_to\_oid/2, 113

- new/1, 114
- snmp\_local\_db*
  - dump/0, 116
  - match/2, 116
  - print/0, 116
  - print/1, 116
  - print/2, 116
  - table\_create/1, 116
  - table\_create\_row/3, 116
  - table\_delete/1, 116
  - table\_delete\_row/2, 116
  - table\_exists/1, 116
  - table\_get\_row/2, 116
- snmp\_mgr*
  - expect/2, 119
  - expect/3, 119
  - expect/4, 119
  - expect/6, 119
  - g/1, 119
  - gb/3, 119
  - gn/0, 120
  - gn/1, 120
  - r/0, 120
  - s/1, 120
  - start/1, 120
  - start\_link/1, 120
  - stop/0, 121
- snmp\_mpd*
  - discarded\_pdu/1, 123
  - generate\_msg/4, 123
  - generate\_response\_msg/4, 122
  - init\_mpd/1, 122
  - process\_packet/4, 122
- snmp\_notification\_mib*
  - configure/1, 124
  - reconfigure/1, 124
- snmp\_pdus*
  - dec\_message/1, 125
  - dec\_message\_only/1, 125
  - dec\_pdu/1, 125
  - dec\_scoped\_pdu/1, 126
  - dec\_scoped\_pdu\_data/1, 126
  - dec\_usm\_security\_parameters/1, 126
  - enc\_encrypted\_scoped\_pdu/1, 126
  - enc\_message/1, 126
  - enc\_message\_only/1, 126
  - enc\_pdu/1, 126
  - enc\_scoped\_pdu/1, 126
  - enc\_usm\_security\_parameters/1, 127
- snmp\_standard\_mib*
  - configure/1, 128
  - inc/1, 128
  - inc/2, 128
  - reconfigure/1, 128
  - reinit/0, 129
  - sys\_up\_time/0, 129
- snmp\_supervisor*
  - start\_master/2, 130
  - start\_master/3, 130
  - start\_sub/0, 130
  - start\_sub/1, 130
  - start\_subagent/3, 131
  - stop\_subagent/1, 131
- snmp\_target\_mib*
  - configure/1, 132
  - reconfigure/1, 132
- snmp\_user\_based\_sm\_mib*
  - configure/1, 133
  - reconfigure/1, 133
- snmp\_view\_based\_acm\_mib*
  - configure/1, 134
  - reconfigure/1, 134
- start/1
  - snmp\_mgr*, 120
- start\_link/1
  - snmp\_mgr*, 120
- start\_master/2
  - snmp\_supervisor*, 130
- start\_master/3
  - snmp\_supervisor*, 130
- start\_sub/0
  - snmp\_supervisor*, 130
- start\_sub/1
  - snmp\_supervisor*, 130
- start\_subagent/3
  - snmp\_supervisor*, 131
- stop/0
  - snmp\_mgr*, 121
- stop\_subagent/1
  - snmp\_supervisor*, 131
- sys\_up\_time/0
  - snmp\_standard\_mib*, 129
- table\_create/1
  - snmp\_local\_db*, 116
- table\_create\_row/3

- snmp\_local\_db* , 116
- table\_delete/1
  - snmp\_local\_db* , 116
- table\_delete\_row/2
  - snmp\_local\_db* , 116
- table\_exists/1
  - snmp\_local\_db* , 116
- table\_func/2
  - snmp\_generic* , 108
- table\_func/4
  - snmp\_generic* , 108
- table\_get\_elements/3
  - snmp\_generic* , 109
- table\_get\_row/2
  - snmp\_local\_db* , 116
- table\_next/2
  - snmp\_generic* , 109
- table\_row\_exists/2
  - snmp\_generic* , 109
- table\_set\_elements/3
  - snmp\_generic* , 109
- universal\_time\_to\_date\_and\_time/1
  - snmp* , 102
- unload\_mibs/2
  - snmp* , 102
- unregister\_subagent/2
  - snmp* , 103
- user\_err/2
  - snmp\_error* , 105
- validate\_date\_and\_time/1
  - snmp* , 103
- variable\_func/2
  - snmp\_generic* , 109
- variable\_func/3
  - snmp\_generic* , 109
- variable\_get/1
  - snmp\_generic* , 109
- variable\_set/2
  - snmp\_generic* , 109