Solaris System Management Agent Developer’s Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 817–3155–10
January 2005

Contents

Tables

Figures

Examples

Preface

This manual, *Solaris System Management Agent Developer's Guide*, describes how to develop MIB modules for use in extending the System Management Agent.

The manual also includes information about migrating existing modules that were developed for the Solstice Enterprise Agents.

Note – This Solaris™ release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems appear in the *Solaris 10 Hardware Compatibility List* at `http://www.sun.com/bigadmin/hcl`. This document cites any implementation differences between the platform types.

In this document the term "x86" refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families. For supported systems, see the *Solaris 10 Hardware Compatibility List*.

Who Should Use This Book

This manual is intended for developers who want to add new management data to the System Management Agent. This data can then be manipulated through network management programs.

The manual assumes that you are familiar with the following technologies:

- C programming concepts
- SNMPv1, SNMPv2c, and SNMPv3 protocols
- Structure of Management Information (SMI) v1 and v2
- Management Information Base (MIB) structure

- Abstract Syntax Notation (ASN.1)

How This Book Is Organized

This manual contains the following chapters:

Chapter 1 provides an introduction to the Simple Network Management Protocol (SNMP) and the System Management Agent (SMA).

Chapter 2 provides basic guidelines for creating System Management Agent modules.

Chapter 3 discusses the handling of data in scalar form and in tables.

Chapter 4 explains how to store module data that is preserved when the agent is restarted.

Chapter 5 explains how to implement alarms in modules.

Chapter 6 discusses the ways to deploy your module, as a subagent or a dynamically loaded module.

Chapter 7 describes how to implement a module to allow more than one instance of the module to run on a host.

Chapter 8 discusses the ways that you can enable a module to collect data over a long period of time.

Chapter 9 describes the Entity MIB and its API functions.

Chapter 10 contains information for developers who want to migrate an SEA subagent from Solstice Enterprise Agents to use in the System Management Agent.

Appendix A lists System Management Agent resources that you might find helpful.

Appendix B lists the MIBs that are included in the System Management Agent.

Glossary contains definitions of terms that are used in this manual.

Related Reading

For general information on SNMP and writing MIBs, you might find the following books helpful:

- *Essential SNMP* by Douglas R. Mauro and Kevin J. Schmidt, published by O'Reilly and Associates.
- *Understanding SNMP MIBs* by David T. Perkins and Evan McGinnis, published by Prentice Hall.

If you intend to use the Entity MIB for the management of hardware, you should read the following RFC:

Internet Engineering Task Force RFC Number 2737 on the Entity MIB at `http://www.ietf.org/rfc/rfc2737.txt`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Ordering Sun Documentation

Sun Microsystems offers select product documentation in print. For a list of documents and how to order them, see "Buy printed documentation" at `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. Perform a *patch analysis*. Do *not* save the file. [Note that some emphasized items appear bold online.]

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction to the System Management Agent

The System Management Agent is a Simple Network Management Protocol (SNMP) agent. This chapter contains the following topics:

Overview of SNMP Agents

SNMP uses the term *manager* for the client application that accesses the data about a managed device or system. The manager usually runs on a system that is different from the managed system. The term *agent* is used for the program that implements the protocol stack for servicing the requests from the manager. The SNMP agent typically runs on the managed device. The agent offers services on a designated TCP/IP port. The default SNMP port is 161.

Information about the target device is contained in a Management Information Base (MIB). MIBs are used by agents and managers so that both programs have knowledge of the data available. The MIB tells the manager about the device's functions and data. The MIB also tells the manager how to address or access that information in the form of managed objects. To access this management information, the manager issues requests to the agent. The requests contain identifiers for the MIB's data objects that are of interest to the manager. If the request can be successfully completed, the agent returns a response that contains the values for the required data objects.

Most SNMP agents support the basic SNMP protocol stack, and some minimal MIBs. However, to make management of a device more effective, additional MIBs must be supported on the managed device. The additional MIBs are provided by device vendors to provide management information about custom features of the managed device.

A MIB that is added to an SNMP agent is commonly known as an *extension* because the MIB extends the capabilities of the agent. An agent that can accept extensions is *extensible*. The System Management Agent (SMA), described in this manual, is an extensible agent. The extensions to the System Management Agent are called extension *modules*.

Overview of the System Management Agent

The System Management Agent (SMA) is an SNMP agent that is based on an open source project, Net-SNMP at `http://www.net-snmp.org`. This base agent supports SNMPv1, v2c, and v3 protocols.

Note – The Net-SNMP version used in SMA is 5.0.9.

The following diagram shows the structure of the Net-SNMP agent, and is followed by an explanation of the components.

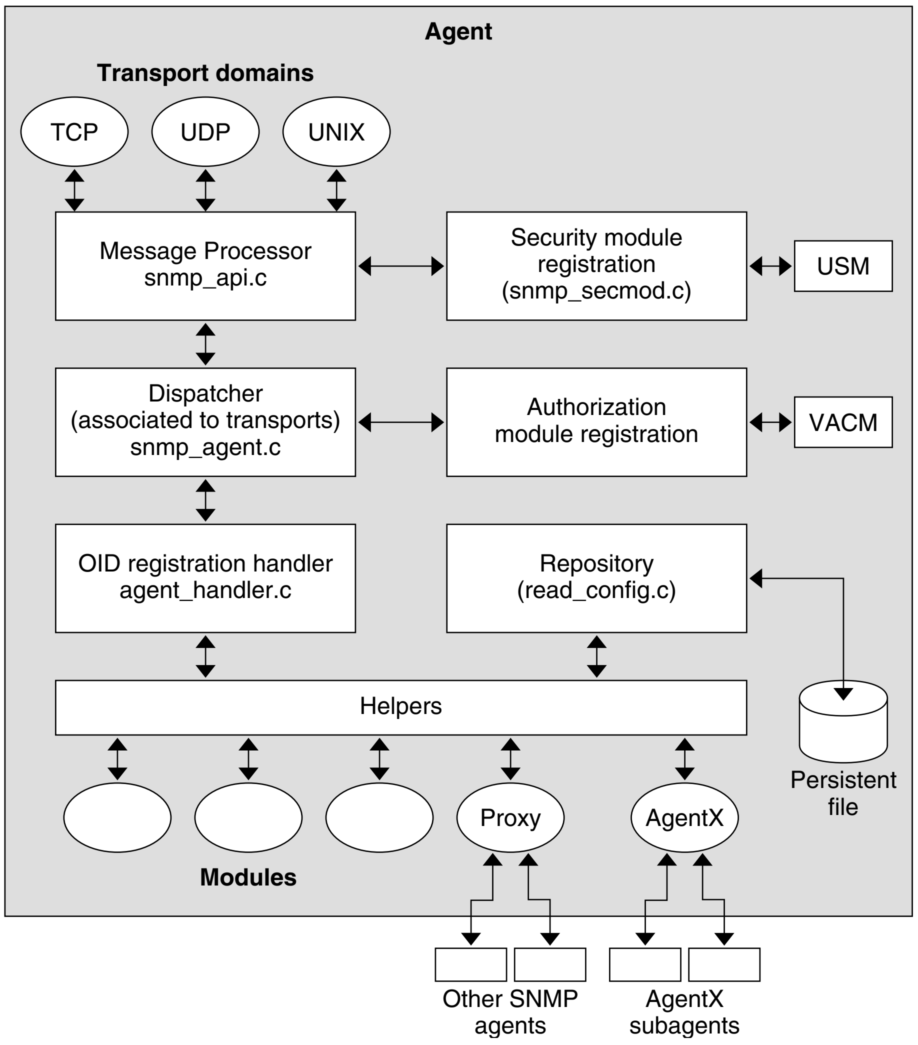
Agent
Transport domains
TCP
UDP
UNIX
Message Processor snmp_api.c
Security module registration (snmp_secmod.c)
USM
Dispatcher (associated to transports) snmp_agent.c
Authorization module registration
VACM
OID registration handler agent_handler.c
Repository (read_config.c)
Helpers
Persistent file
Proxy
AgentX
Modules
Other SNMP agents
AgentX subagents

FIGURE 1–1 Net-SNMP Architecture

The components of the Net-SNMP agent are:

- Transport domains

  The Net-SNMP agent currently supports the transports TCP, UDP, and UNIX Domain Sockets. The agent can receive and transmit SNMP messages through these transports. The agent's implementation of each transport implements functions to send and receive raw SNMP data. The raw messages received by the transport domains are passed to the message processor for further processing. The message processor also transfers raw SNMP messages to the transport domain for sending.

- Message processor

  The message processor decodes raw SNMP messages into internal PDU structures. The processor also encodes PDUs into raw SNMP messages. The SNMP messages are encoded by using Binary Encoding Rules, which are described in RFC 3416 and RFC 1157. The message processor also handles the security parameters in the SNMP messages. If the messages include User-based Security Model (USM) security parameters, the message processor passes the required parameters to the USM module. Additionally, trap messages from modules can be sent to the message processor for transmission.
- USM module

  The USM module handles all processing that is required by the User-based Security Model as defined in RFC 3414. The module also implements the SNMP-USER-BASED-SM-MIB as defined in the same RFC. The USM module, when initialized, registers with the agent infrastructure. The message processor invokes the USM module through this registration.

  The USM module decrypts incoming messages. The module then verifies authentication data and creates the PDUs. For outgoing messages, the USM module encrypts PDUs and generates authentication data. The module then passes the PDUs to the message processor, which then invokes the dispatcher.

  The USM module's implementation of the SNMP-USER-BASED-SM-MIB enables the SNMP manager to issue commands to manage users and security keys. The MIB also enables the agent to ensure that a requesting user exists and has the proper authentication information. When authentication is done, the request is carried out by the agent.

  The various keys that the USM module needs to perform encryption and authentication operations are stored persistently.

  See Chapter 4, "Managing Security," in *Solaris System Management Agent Administration Guide* for more information about USM.
- Dispatcher

  The dispatcher is responsible for routing messages to appropriate destinations. After the USM module processes an incoming message into PDUs, the dispatcher performs an authorization check. This authorization check is done by the registered access control module, which is the VACM module. If the check succeeds, the dispatcher uses the agent registry to determine the module that has registered for the relevant object identifier. The dispatcher then invokes appropriate operations on the module. A particular request might cause the dispatcher to invoke several modules if the SNMP request contains multiple variables. The dispatcher keeps track of outstanding requests through session objects. Responses from the modules are then dispatched to the transports that are associated with the session objects. The message processor performs the appropriate message encoding.
- VACM module

  The View-based Access Control Model is described in RFC 3415. This RFC also defines the SNMP-VIEW-BASED-ACM-MIB. The MIB specifies objects that are needed to control access to all MIB data that is accessible through the SNMP agent.

  Upon initialization, the VACM module registers as the access control module with the agent infrastructure. The VACM module implements access control checks according to several parameters that are derived from the SNMP message. These parameters specify:

  - The security model being used, which can be USM, v1 communities, or v2 communities
  - The security name, which is user name in USM, and community string in v1 and v2
  - The context
  - The object being accessed
  - The operation being performed

  By implementing the SNMP-VIEW-BASED-ACM-MIB, the VACM module handles manipulation of various table entries that are mandated by VACM. These table entries are looked up in performing the VACM check and are maintained persistently in the agent configuration file.

  See Chapter 4, "Managing Security," in *Solaris System Management Agent Administration Guide* for more information about VACM.

- Repository

  The agent configuration file, `snmpd.conf`, is the repository for the agent. Configuration tokens for various modules are stored in the repository. The modules have access to these configuration tokens when the modules are initialized. Modules can also register callback routines with the repository. The callbacks are invoked when the module state needs to be persisted, or written to disk to be retrieved later. Within the callbacks, each module is allowed to output its state. When the agent shuts down, the callbacks are used to save the modules' state. An SNMP SET command can also be used to cause a module's state to be persisted.

- OID registration handler

  The OID registration handler, or agent registry, handles the registration of object identifiers that are specified by modules.

- Proxy module

  The proxy module handles proxy forwarding of SNMP messages to and from other SNMP agents. The proxy module can also map between SNMPv1 and SNMPv2 protocols according to the rules specified in RFC 2576.

  The proxy module stores its configuration tokens in the agent configuration file. A particular configuration entry can associate OIDs within a context with another SNMP agent. The configuration file also specifies community strings and destination transport end points. By using these configuration tokens, the proxy module registers as the handler for the specific OIDs. When an incoming request for any of the proxy module's OIDs reaches the dispatcher, the dispatcher invokes the proxy module. The proxy module then issues appropriate SNMP requests to the target agents. Responses are returned back to the dispatcher.

The System Management Agent uses the proxy module for interaction with the Solstice Enterprise Agents.

- AgentX module

  The AgentX module implements RFCs 2741 and 2742. The AgentX module registers as the handler for the AgentX-related registration tables defined in the AGENTX-MIB. The transports that are used for the AgentX protocol interactions are specified in the agent configuration file. In the Net-SNMP agent, the transports are typically UNIX domain sockets. When the AgentX module is initialized, the module creates sessions on these transports and registers as the handler for these sessions. In the System Management Agent, the only allowable AgentX transport is UNIX domain sockets, so only sessions on UNIX domain sockets are created.

  When an AgentX subagent starts, the subagent sends its registration requests with messages that use the AgentX protocol to the master agent. The requests are received by the sessions that have been created by the AgentX module. The message processor decodes the message, then invokes the AgentX module. The AgentX module, rather than the dispatcher module, is the handler for these sessions.

  The AgentX module then registers as the handler for OIDs that are specified in the subagent registration message. When requests for these OIDs are received by the dispatcher, the requests are directed to the AgentX module, which in turn connects to the required subagent. Requests to unregister OIDs are handled similarly.

- Extension modules

  Extension modules, which are depicted at the bottom of Figure 1–1, are the means by which MIBs are implemented in the agent. An extension module registers with the agent all the object identifiers that the module manages. The module also implements functionality to perform SNMP operations on the module's objects. As shown in Figure 1–1, helper routines or handlers in the API can be inserted between a module and the agent infrastructure. These handlers can have various functions, such as handling details of table iterations or providing debug output.

Extending the Agent

The Net-SNMP agent can be extended in the following ways:

- The agent can be recompiled to include a static module, which becomes part of the agent infrastructure. Static modules are initialized on agent start up. Examples of static modules include the VACM and USM modules. The System Management Agent was developed by compiling several static modules for MIBs that are not included in the Net-SNMP agent. See "Features Added in System Management Agent" on page 23 for a list of the MIBs included in SMA.

  You cannot deploy your own modules as static modules with SMA because you cannot recompile the SMA code.

- Modules can be loaded into the master agent's process image. A shared object is dynamically loaded into the agent when the agent is running. The shared object registers the OIDs for the MIB that is supported by the shared object. The location of the shared object libraries for the module can be specified through SNMP requests or in the agent configuration file.

- Modules can be loaded into secondary SNMP *subagents*. Subagents are separate executable programs that can dynamically register themselves with the agent that is running on the designated SNMP port. The monitoring agent processes any SNMP request that comes to the SNMP port, and can send a request to a subagent, if needed. In this scenario, the agent on the designated port is called the *master agent*. The AgentX RFCs 2741 and 2742 define the protocols between the subagent and master agent as well as the MIBs that contain details of the registrations. For more information on master agents and subagents, see Chapter 6.
- A module can be delivered as an SNMP agent. The master agent can interact with such agents through a proxy mechanism.

Note – The System Management Agent supports module deployment in the form of dynamically loaded modules or subagents. The agent also provides a proxy mechanism. You cannot recompile the System Management Agent.

See Chapter 6 for information about how to deploy modules as dynamic modules and in subagents.

Features Added in System Management Agent

The SMA product includes the Net-SNMP agent, Net-SNMP developer tools, and Net-SNMP API libraries that enable the agent to be extended through modules. You can use the tools and APIs to create a module to support the custom features of a managed device. A management program can then be used for monitoring and managing the device.

The SMA extends the Net-SNMP agent to support the following MIBs:

- MIB II, described in `http://www.ietf.org/rfc/rfc1213.txt`, defines the structure of management information and the network management protocol for TCP/IP-based networks. The SMA implements all the object groups of MIB II except the EGP group.
- Host Resources MIB, described in `http://www.ietf.org/rfc/rfc2790.txt`, defines the structure of management information for managing host systems. The SMA implements the same host resources MIB that is included in the base Net-SNMP agent.
- Sun MIB is the MIB II with Sun-specific object groups added. Sun MIB was originally provided in the Solstice Enterprise Agents product beginning with the Solaris 2.6 operating system. The SMA implements the following groups from the Sun MIB:
  - Sun System group

- Sun Processes group
- Sun Host Performance group

Support for these MIBs is provided as static modules that run in the SMA agent. The agent was created by recompiling the Net-SNMP agent to include these modules.

The SMA provides an additional MIB for hardware, the Entity MIB, in an external dynamically loaded module. The Entity MIB is described in RFC 2737 at `http://www.ietf.org/rfc/rfc2737.txt`. The Entity MIB is implemented as a skeleton MIB, so that module developers can populate the various tables of the MIB. The Entity MIB can be used by a single agent for managing multiple logical entities and physical entities. For more information about the Entity MIB, see Chapter 9.

Contents of the SMA for Developers

SMA includes the following content for developers:

- Developer tools, and Perl modules needed by the tools
- API libraries for using Net-SNMP functions
- API library for using the Entity MIB functions
- Demo modules, for demonstrating how to implement some types of data modeling

In addition, you can install the `SUNWsmaS` package, which contains the source code for Net-SNMP. See the *Solaris System Management Agent Administration Guide* for installation instructions.

File Locations of Developer Files

The developer files are installed in the locations that are shown in the following table.

TABLE 1–1 File Locations for Developer Content

Directory	Developer Content
`/usr/demo/sma_snmp`	Sample modules for demonstration purposes. See "Demonstration Modules" on page 26 for more information.
`/usr/sfw/bin`	Command line tools that are useful for developers. For more information on these tools, see "SMA Tools" on page 25.

TABLE 1–1 File Locations for Developer Content *(Continued)*

Directory	Developer Content
`/usr/sfw/sbin`	Executable files for the `snmpd` agent daemon and `snmptrapd` trap daemon, which provide the SNMP services.
`/usr/sfw/lib`	The 32-bit shared libraries that contain the API functions from Net-SNMP, and the `libentity.so` library, which defines functions for using the Entity MIB. This directory is supplied on all Solaris platforms. See "API Libraries" on page 26 for more information.
`/usr/sfw/lib/sparcv9`	The 64-bit shared libraries that contain the API functions from Net-SNMP, and the `libentity.so` library, which defines functions for using the Entity MIB. This directory is supplied only on 64-bit Solaris on SPARC® platforms. See "API Libraries" on page 26 for more information.
`/usr/sfw/include`	Header files needed by API libraries.
`/usr/sfw/doc/sma_snmp/html`	HTML documentation for Net-SNMP API functions.
`/etc/sma/snmp`	Configuration files that are used by the `mib2c` tool.
`/etc/sma/snmp/mibs`	The MIBs supported by the System Management Agent.
`/usr/perl5/vendor_perl/5.8.3/sun4-solaris-64int`	Perl modules needed by the `mib2c` tool.
`/usr/share/sma_snmp`	Source code for Net-SNMP. The code is provided in the `SUNWsmaS` package, which is not installed by default during Solaris installation. The package must be installed manually from the Solaris media. For instructions for installing the `SUNWsmaS` package, see the *Solaris System Management Agent Administration Guide*.

SMA Tools

The SMA includes many command-line tools, which are described in the `sma_snmp`(5) man page.

Each tool has an associated man page. Links to all the man pages for the product are included in Appendix A. The tools are located in `/usr/sfw/bin`.

The snmp commands can be used to query the agent to test your modules. Read the man pages for detailed usage information.

API Libraries

The following API libraries are included with the SMA product:

- libnetsnmp
- libnetsnmpagent
- libnetsnmpmibs
- libnetsnmphelpers
- libentity

The libentity library is not part of Net-SNMP, but is a customization for the SMA product.

On SPARC platforms, the 32–bit Net-SNMP libraries are contained in the /usr/sfw/lib directory. The 64–bit Net-SNMP libraries are contained in the /usr/sfw/lib/sparcv9 subdirectory.

On x86 platforms, only the 32–bit Net-SNMP libraries are available in the /usr/sfw/lib directory.

The functions contained in the Net-SNMP libraries are used in the MIB modules that you create, as well as in the agent. Documentation from Net-SNMP for using the API functions is contained in /usr/sfw/doc/sma_snmp/html.

The SMA includes the same Net-SNMP API functions that are available with the open source Net-SNMP agent. "API Functions" on page 136 includes a list of functions that are certified to work with the System Management Agent.

Demonstration Modules

The /usr/demo/sma_snmp directory contains several demonstration modules. The demo modules illustrate methods for creating modules to solve various kinds of information-gathering problems. Later chapters in this manual discuss the demo modules in detail. The following table lists and describes the demo modules. The table also provides cross-references to the sections that discuss the demos.

TABLE 1–2 Descriptions of Demonstration Modules

Module Name	Demonstrates	Discussed in Section
demo_module_1	Data modeling for scalar objects	"Scalar Objects" on page 41

TABLE 1–2 Descriptions of Demonstration Modules *(Continued)*

Module Name	Demonstrates	Discussed in Section
demo_module_2	Data modeling for a simple table with writable objects	"Simple Tables" on page 43
demo_module_3	Data modeling for a general table	"General Tables" on page 48
demo_module_4	Implementing alarms	"demo_module_4 Code Example for Alarms" on page 61
demo_module_5	Persistence of module data across agent restarts	"demo_module_5 Code Example for Persistent Data" on page 54
demo_module_6	Running multiple instances of a module on a single host	"Implementing Multiple Instances of a Module" on page 75
demo_module_7	Dynamically updating multi-instance modules	"Enabling Dynamic Updates to a Multiple Instance Module" on page 78
demo_module_8	Implementing a module as an AgentX subagent	"Deploying a Module as a Subagent" on page 71
demo_module_9	Implementing objects that wait for external events without blocking the agent	"SNMP Alarm Method for Data Collection" on page 86
demo_module_10	Module design that handles long running data collections so that their values can be polled by an SNMP manager	"SNMP Manager Polling Method for Data Collection" on page 88
demo_module_11	How to use the Entity MIB API functions	"SMA Entity MIB Implementation" on page 93
demo_module_12	How use Solstice Enterprise Agents code templates and SMA code templates to help re-implement Solstice Enterprise Agents subagents as SMA modules	"Migrating Solstice Enterprise Agent Subagents to SMA" on page 129

Technical Support for Developers

Technical support for developers of modules for the System Management Agent is provided through the Net-SNMP open source community at `http://www.net-snmp.org`. You might find the developers discussion mailing list `net-snmp-coders@lists.sourceforge.net` to be helpful. An archive for the mailing list is located at `http://sourceforge.net/mailarchive/forum.php?forum_id=7152`.

CHAPTER 2

Creating Modules

This chapter provides basic guidelines for creating System Management Agent modules. The chapter includes a process you can use to implement a MIB as a module in System Management Agent. Guidelines for naming components of your implementation to avoid conflicts are also included.

The following topics are discussed:

About Modules

The term *module* as used in this manual has two closely related meanings. Module refers generically to the "container" of the new pieces of management data that the developer needs to inform the agent about. In this sense, a module is an abstract concept.

However, an abstract module must be represented as a shared object file, which runs on a managed system. The shared object file, or the associated program, is often referred to as a module. Therefore, a module can be defined as a C program that works with the SMA to manage additional resources.

All modules communicate through the API library functions. The API functions are used whether the modules are compiled in the agent, or loaded dynamically, or running in a separate subagent.

Overview of Creating Modules

You can create modules for the System Management Agent to allow a specific application, device, system, or network to be managed through a management application. The System Management Agent includes and documents the functions that are required by a module. The functions are used to register a module with the agent, to handle requests for module data, and to perform other module tasks.

You are not required to code a module manually, although you can if you prefer. Refer to the `http://www.net-snmp.org/` web site for information about writing a module manually. The process is outside the scope of this document.

The high-level process described in this manual for implementing a module is as follows:

1. Define the MIB for the objects to be managed.

   To define a MIB, you must know what management data is associated with the system or entity to be managed. You must assign variable names to each discrete management item. You must also determine the attributes and ASN.1 data types. MIB definition is outside the scope of this manual. See "Defining a MIB" on page 31 for more information about MIBs.

2. Generate code templates for a module from the MIB.

   To generate code templates, you convert the MIB nodes into C source code files with the `mib2c` tool. The code templates include API functions for registering the data, and handling the requests for the data. See "Generating Code Templates" on page 33 for more information.

3. Modify the code templates to fill in the data collection and management portions of the module.

   To modify the code templates, you must determine how to implement much of the functionality of the agent. See "Modifying Code Templates" on page 35 for more information.

4. Compile the C files into a shared object file.

   You compile a module for the System Management Agent as you would compile any C shared object file.

5. Decide on the deployment method and configuration of the module.

   You must determine whether to configure the module as a separate subagent, or to load the module dynamically into the SNMP agent. See Chapter 6 for information about deployment.

Defining a MIB

MIB definition is one of the more time-consuming steps of creating a module. MIB syntax can be quite complicated, and is outside the scope of this document. Refer to "Related Reading" on page 15 for suggestions of other sources of information about MIB syntax.

Tip – The `mib2c` tool, used for converting MIBs to C code, includes error checking for MIB syntax. You can use `mib2c` to check your MIB syntax as you create your MIB, even before you are ready to convert the MIB.

You should consider using one of the standard MIBs that are included with the SMA as a model for creating your MIB. The `/etc/sma/snmp/mibs` directory contains all the standard MIBs supported by the SMA.

The following MIBs can be used as examples to emulate because the MIBs have been found to work well with `mib2c`:

- `UCD-DLMOD-MIB.txt`
- `SUN-SEA-EXTENSIONS-MIB.txt`
- `IP-MIB.txt`

Tip – Pay particular attention to the name that is assigned for the `MODULE-IDENTITY`. This name should be equal to the MIB file name with the hyphens removed, and in mixed case. For example, `SUN-SEA-EXTENSIONS-MIB.txt` uses `sunSeaExtensionsMIB` for the `MODULE-IDENTITY`. A MIB file that does not use this format might not work with `mib2c`.

The file `NET-SNMP-EXAMPLES-MIB.txt` is also included in the `/etc/sma/snmp/mibs` directory, and might be helpful in explaining how to define a variety of MIB variable types.

MIB File Names

You must ensure unique names for your MIB files. All custom MIBs to be used with SMA are in the same namespace as the standard MIBs, even if you keep the custom MIBs in a separate directory. Most of the MIBs derived from RFCs have RFC numbers in their names to clearly identify the MIBs, and ensure unique names. Other MIBs follow naming conventions, which decrease the chances of duplicate names.

MIBs are usually named with the following conventions:

- Use all uppercase letters, and use hyphens to separate segments of the file name.
- Begin the MIB name with your company name. For example, if the MIB is for a company that is named Acme, the first segment of the MIB name might be ACME.
- Indicate the type of objects in the middle of the name. For example, if the MIB is for a router, you could use ROUTER in the middle of the name.
- Include `MIB` as the last segment of the name.
- Append a `.txt` file extension.

A sample name that uses these conventions is `ACME-ROUTER-MIB.txt`.

Setting MIB Environment Variables

You should set the `MIBS` and `MIBDIRS` environment variables to ensure that the tools that use the MIBs can find and load your MIB file. Tools that use the MIBs include `mib2c` and all the `snmp` commands such as `snmpget`, `snmpwalk`, and `snmpset`.

Set the `MIBS` environment variable to include the MIB file that you are using. For example, to add a MIB called `MYTESTMIB.txt` to the list of MIBs, use one of the following commands:

In the `csh` or `tcsh` shells:

```
% setenv MIBS +MYTESTMIB
```

In the `sh` or `bash` shells:

```
# MIBS=+MYTESTMIB;export MIBS
```

These commands add your MIB to the list of default MIB modules that the agent supports.

The default search path for MIB files is `/etc/sma/snmp/mibs`. You can modify the MIB search path by setting the `MIBDIRS` variable. For example, to add the path `/home/mydir/mibs` to the MIB search path, type the following commands:

In the `csh` or `tcsh` shells:

```
% setenv MIBDIRS /home/mydir/mibs:/etc/sma/snmp/mibs
% setenv MIBS ALL
```

In the `sh` or `bash` shells:

```
# MIBDIRS=/home/mydir/mibs:/etc/sma/snmp/mibs
# export MIBDIRS
# MIBS=ALL;export MIBS
```

Setting `MIBS` to `ALL` ensures that `mib2c` finds all MIBs in the search location for MIB files. Both the MIB files to be loaded and the MIB file search location can also be configured in the `snmp.conf` file. See the `snmp.conf`(4)man page for more information.

Note – You should avoid copying your MIBs into the `/etc/sma/snmp/mibs` directory. That directory should be reserved for the MIBs provided with SMA.

Generating Code Templates

You use the `mib2c` tool to generate C header files and implementation files from your MIB. You can use the generated C files as templates for your module. You can modify the templates appropriately for your purposes, and then use the templates to make your module. Before the file generation begins, `mib2c` tests your MIB node for syntax errors. Any errors are reported to standard output. You must fix any syntax errors before the code can be generated. This error-checking ability enables you to use `mib2c` as you create your MIB to ensure that the MIB syntax is correct.

Note – Be sure to set your MIB environment variables as described in "Setting MIB Environment Variables" on page 32 before you use `mib2c`.

The `mib2c` command must be run against nodes in the MIB, not on the entire MIB at once. You do not need to specify the MIB name, but the MIB file must be located in a directory on your MIB search path. On the `mib2c` command line, you must specify a configuration file and the name of one or more MIB nodes. The configuration file must matches the type of data in the MIB node. The command must use the following format:

`mib2c -c` *configfile MIBnode [MIBnode2 MIBnode3 ...]*

For example, if you have one node that is called `scalarGroup` in your MIB, you could use the following command to generate the code templates:

```
% mib2c -c mib2c.scalar.conf scalarGroup
```

The files `scalarGroup.h` and `scalarGroup.c` are generated.

If your MIB contains both scalar and table data, you should run `mib2c` separately on the MIB nodes for each type of data. You specify the appropriate configuration file for each type of data.

The following table lists the mib2c configuration files. The table describes the purpose of each configuration file, to help you decide which configuration file to use for your data.

TABLE 2–1 Configuration Files for Use With mib2c Tool

mib2c Configuration File	Purpose
mib2c.scalar.conf	For scalar data, including integers and non-integers. This configuration file causes mib2c to generate handlers for the scalar objects in the specified MIB node. Internal structural definitions, table objects, and notifications in the MIB are ignored.
mib2c.int_watch.conf	For scalar integers only. When you use this configuration file, mib2c generates code to map integer type scalar MIB objects to C variables. GET or SET requests on MIB objects subsequently have the effect of getting and setting the corresponding C variables in the module automatically. This feature might be useful if you want to watch, or monitor, the values of certain objects.
mib2c.iterate.conf	For tables of data that are not kept in the agent's memory. The tables are located externally, and the tables need to be searched to find the correct row. When you use this configuration file, mib2c generates a pair of routines that can iterate through the table. The routines can be used to select the appropriate row for any given request. The row is then passed to the single table handler routine. This routine handles the rest of the processing for all of the column objects, for both GET and SET requests.
mib2c.create-dataset.conf	For tables of data that are kept in the agent's memory. The table does not need to be searched to find the correct row. This configuration file causes mib2c to generate a single handler routine for each table. Most of the processing is handled internally within the agent, so this handler routine is only needed if particular column objects require special processing.
mib2c.array-user.conf	For tables of data that are kept in the agent's memory. The data can be sorted by the table index. This configuration file causes mib2c to generate a series of separate routines to handle different aspects of processing the request. As with the mib2c.create-dataset.conf file, much of the processing is handled internally in the agent. Many of the generated routines can be deleted if the relevant objects do not need special processing.
mib2c.column_defines.conf	To create a header file that contains a #define for each column number in a MIB table.

TABLE 2–1 Configuration Files for Use With `mib2c` Tool *(Continued)*

`mib2c` Configuration File	Purpose
`mib2c.column_enums.conf`	To create a header file that contains a `#define` for each enum of common values used by the columns in a MIB table.

The `mib2c`(1M) man page includes more details about using the `mib2c` tool. You should also see Chapter 3 for more examples of using `mib2c`.

Modifying Code Templates

The code templates that are generated by `mib2c` include code that registers the OIDs for the MIB data and handles the requests for the data. The `init_`*module* routine in the *mibnode*`.c` template provides the basic code for data retrieval. You must modify the templates to provide the data collection and data management, or instrumentation, of your module. See "`init_`*module* Routine" on page 39 for information about modifying the initialization routine.

The following table shows where to find more information about how to do various types of data collection.

TABLE 2–2 Data Collection Documentation

Type of data	Reference
Scalar objects	"Scalar Objects" on page 41
Simple tables	"Simple Tables" on page 43
General tables	"General Tables" on page 48
Long running	Chapter 8

Configuring the Module

Configuration of the module depends partly on the module. You can provide automatic configuration as part of the installation process for your module. Alternatively, you can provide the steps and suggestions as part of the end user documentation. If you want users to be able to set configuration parameters for your module, you can store configuration parameters in a configuration file. The parameters can then be retrieved by the module whenever the module starts. See Chapter 4 for information.

For any module, you must decide whether to run the module as a subagent or a dynamically loading module. See Chapter 6 for more information.

Delivering the Module

When the module code is complete, you must decide how to deliver the module. If you are creating a module that must be distributed and then be installed, you should use the operating system's native software delivery model. For the Solaris operating system, you should use packages as described in the *Application Packaging Developer's Guide*.

Namespace Issues

This section explains the naming conventions for the System Management Agent. The conventions are required to enable all developers to avoid namespace collisions.

Avoiding Namespace Collisions

Namespace is a term used to indicate the complete set of possible names that can exist together in a certain "space." Namespaces exist in the computer world and in the real world. For example, the names of people in a group, such as the passenger list in an aircraft, forms a namespace. In the computer world, a namespace might be a list of file names in a directory, or the function names in a source code file.

A namespace usually requires names to be unique, to ease the addressing of an individual entity. In the real world, the names of entities in a namespace might not always be unique. For example, there might be two aircraft passengers with the same name. In such situations, an attribute other than the name of the entities of the namespace must be used. For example, the seat numbers might form the namespace of the passengers on the aircraft.

The namespaces in the computing world mandate that uniqueness is ensured. For example, you must have unique names for all the files in a directory or functions that are part of the same program.

Namespace collision occurs if parts of the namespace delivered by different people have the same names. For example, two vendors might come up with the same library name and install in the same directory. A recent trend is to make the directory name part of the namespace, to ensure different directories for different vendors or different products. Even if the file names are the same, the file names are in different directories.

For the SMA developer, several areas are susceptible to namespace collisions. The following sections discuss naming conventions that you must follow to greatly reduce the possibility of having naming issues.

Module Names

The module name should be based closely on the name of the MIB that is implemented by the module. MIB name guidelines are discussed in "MIB File Names" on page 31.

Use the following guidelines to name your module, beginning with the name of the MIB file:

- Remove the hyphens
- Remove the word MIB
- Remove the `.txt`
- Convert to lowercase

For example, if your MIB name is `ACME-ROUTER-MIB.txt`, you should name the module `acmerouter`. When you compile, the shared object that results is `acmerouter.so`.

Library Names

You must ensure unique names for your custom libraries because all libraries to be used with SMA are delivered into a single `lib` namespace. You should observe the following guidelines in naming your libraries:

- Observe the guidelines for creating unique MIB names in "MIB File Names" on page 31.
- Observe the guidelines for naming your module in "Module Names" on page 37.
- Add the prefix `lib` to your module name to create the name of your library.

For example, assume that your MIB name is `ACME-ROUTER-MIB.txt`. Your module name is `acmerouter`. The associated library should be named `libacmerouter.so`. The `.so` extension is added when you compile.

CHAPTER 3

Data Modeling

This chapter provides information on how to modify the `init_`*module*() routine of a module to handle various types of data. The chapter discusses the related code examples that are provided with the System Management Agent:

`demo_module_1`	Scalar data example
`demo_module_2`	Simple table example
`demo_module_3`	General table example

The chapter includes the following topics:

`init_`*module* Routine

When a module is loaded in the agent, the agent calls the `init_`*module*() routine for the module. The `init_`*module*() routine registers the OIDs for the objects that the module handles. After this registration occurs, the agent associates the module name with the registered OIDs. All modules must have this `init_`*module*() routine.

The `mib2c` utility creates the `init_`*module*() routine for you. The routine provides the basic code for data retrieval, which you must modify appropriately for the type of data.

If you have several MIB nodes in your MIB, the `mib2c` utility creates several `.c` files. Each generated file contains an `init_`*mibnode*() routine. A module must have only one initialization routine, which must conform to the convention of `init_`*module*(). Therefore, when you have more than one MIB node represented in your module, you must combine the initialization content of all the generated `.c` files into one file to ensure that the initialization routine for each MIB node is called by `init_`*module*().

You can combine files to build a module in one of the following ways:

- Create a module file to call all the initialization routines.

  With this approach, the routine `init_myMib()` in `myMib.c` might look similar to the following pseudo code:

  ```
  #include "scalarGroup.h"
  #include "tableGroup.h"
  ...
      init_myMib() {

          init_scalarGroup();
          init_tableGroup();
      }
  ```

  where `init_scalarGroup()` and `init_tableGroup()` are in different files.

- Combine the initialization routines' code into one initialization routine.

  If you used this approach, the routine `init_myMib()` might be similar to the following pseudo code:

  ```
  init_myMib() {

            <init code - scalarGroup>  /* found in scalarGroup.c */
            <init code - tableGroup>  /* found in tableGroup.c */
          }
  ```

In both cases, the rest of the code in `myMib.c` might be similar to the following pseudo code:

```
/* get/set handlers for scalarGroup found in scalarGroup.c */

/* get_first/get_next/handler for tableGroup - found in tableGroup.c */
```

The following sections discuss how the data retrieval code must be modified in your module for different types of data.

Scalar Objects

Scalar objects are used for singular variables that are not part of a table or an array. If your MIB contains scalar objects, you must run `mib2c` with a scalar-specific configuration file on the MIB nodes that contain the scalars. You should use the following command, where *mibnode1* and *mibnode2* are top-level nodes of scalar data for which you want to generate code:

```
mib2c -c mib2c.scalar.conf mibnode1 mibnode2 ...
```

You can specify as many nodes of scalar data as you want. This command generates two C code files that are named *mibnode*`.c` and *mibnode*`.h` for each MIB node that is specified in the command line. You must modify the *mibnode1*`.c` and *mibnode2*`.c` files to enable the agent to retrieve data from scalar objects. See the `mib2c`(1M) man page for more information about using the `mib2c` tool.

Now, compile the MIB and example code as described in "`demo_module_1` Code Example for Scalar Objects" on page 41.

`demo_module_1` Code Example for Scalar Objects

The `demo_module_1` code example is provided to help you understand how to modify the code generated by the `mib2c` command to perform a scalar data retrieval. The `demo_module_1` code example is located by default in the directory `/usr/demo/sma_snmp/demo_module_1`.

The `README_demo_module_1` file contains instructions that describe how to perform the following tasks:

- Generate code templates from a MIB that contains scalar objects
- Compile source files to generate a shared library object that implements a module
- Set up the agent to dynamically load the module
- Test the module with `snmp` commands to show that the module is functioning as expected

The `demo_module_1` is set up to allow you to generate code templates `me1LoadGroup.c` and `me1LoadGroup.h`. You can then compare the generated files to the files `demo_module_1.c` and `demo_module_1.h`. The `mib2c` utility generates `me1LoadGroup.c`, which contains the `init_me1LoadGroup()` function. You should compare this function to the `init_demo_module_1()` function in the `demo_module_1.c` file.

The `demo_module_1.c` and `demo_module_1.h` files have been modified appropriately to retrieve scalar data. You can use these files as a model for learning how to work with scalar data in your own module. The instructions then explain how to compile the modified source files to create a functioning module.

Modifications for Scalar Data Retrieval

The `demo_module_1` example code, `demo_module_1.c`, provides the system load average for 1, 5 and 15 minutes, respectively.

The `init_demo_module_1()` function call defines the OIDs for the following three scalar objects:

- `me1SystemLoadAvg1min`
- `me1SystemLoadAvg5min`
- `me1SystemLoadAvg15min`

These OIDs are set up in the `demo_module_1.c` source file, to reflect what is in the `SDK-DEMO1-MIB.txt`. The OIDs are defined as follows:

```
static oid me1SystemLoadAvg15min_oid[] =
  { 1,3,6,1,4,1,42,2,2,4,4,1,1,3, 0 };
static oid me1SystemLoadAvg1min_oid[] =
  { 1,3,6,1,4,1,42,2,2,4,4,1,1,1, 0 };
static oid me1SystemLoadAvg5min_oid[] =
  { 1,3,6,1,4,1,42,2,2,4,4,1,1,2, 0};
```

The `mib2c` command used the `netsnmp_register_read_only_instance()` function to register these handler functions:

- `get_me1SystemLoadAvg1min()`
- `get_me1SystemLoadAvg5min()`
- `get_me1SystemLoadAvg15min()`

In this way, when a GET or GET_NEXT request is received, the corresponding handler function is called.

For example, for the 15 minute load average, you can manually register the `get_me1SystemLoadAvg15min()` handler function. The handler retrieves data on the `me1SystemLoadAvg15min` scalar. You must place the handler in the `netsnmp_register_read_only_instance()` function as follows:

```
netsnmp_register_read_only_instance
(netsnmp_create_handler_registration
("me1SystemLoadAvg15min",
get_me1SystemLoadAvg15min,
me1SystemLoadAvg15min_oid,
OID_LENGTH(me1SystemLoadAvg15min_oid),
HANDLER_CAN_RONLY));
```

Alternatively, you can use the `mib2c` command to generate the function bodies of the handler functions for you. Replace */* XXX...* in the generated code with your own data structure for returning the data to the requests. For instance, the following code must be modified:

```
case MODE_GET:
snmp_set_var_typed_value(requests->requestvb, ASN_OCTET_STR, (u_char
 *) /* XXX: a pointer to the scalar's data */,
 /* XXX: the length of the data in bytes */);
break;
```

This code must be modified to include your own data structure for returning data to the requests. Replace the */* XXX...* that is shown in the preceding code.

```
case MODE_GET:
data = getLoadAvg(LOADAVG_1MIN);
snmp_set_var_typed_value(requests->requestvb, ASN_OCTET_STR, (u_char
 *) data , strlen(data));
free(data);
break;
```

Note that the input MIB file contains the specification of a table as well as scalar data. When you run `mib2c -c mib2c.scalar.conf` *scalar-node* the template code is generated only for the scalar nodes in the MIB.

Simple Tables

A simple table has four characteristics:

- The table is indexed by a single integer value
- Such indexes run from 1 to a determinable maximum
- All indexes within this range are valid
- The data for a particular index can be retrieved directly by, for example, indexing into an underlying data structure

If any of these conditions are not met, the table is not a simple table but a *general* table. The techniques described here are applicable only to simple tables.

Note – `mib2c` assumes that all tables are simple. For information on handling the general tables case, see "General Tables" on page 48.

If your MIB contains simple tables, you must run `mib2c` with a configuration file that handles code generation for simple tables. You should use the following command, where *mibnode1* and *mibnode2* are top level nodes of tabular data for which you want to generate code:

```
mib2c -c mib2c.iterate.conf mibnode1 mibnode2 ...
```

You can specify as many nodes of simple table data as you want. This command generates two C code files that are named *mibnode*`.c` and *mibnode*`.h` for each MIB node that is specified in the command line. You must modify the *mibnode1*`.c` and *mibnode2*`.c` files to enable the agent to retrieve data from simple tables. See the `mib2c`(1M) man page for more information about using the `mib2c` tool.

The `demo_module_2` code example shows how to generate code templates for simple tables.

`demo_module_2` Code Example for Simple Tables

The `demo_module_2` code example is provided to help you understand how to modify the code generated by the `mib2c` command to perform data retrieval from simple tables. The `demo_module_2` code example is located by default in the directory `/usr/demo/sma_snmp/demo_module_2`.

The `README_demo_module_2` file contains instructions that describe how to do the following tasks:

- Generate code templates from a MIB that contains a simple table
- Compile source files to generate a shared library object that implements a module
- Set up the agent to dynamically load the module
- Test the module with `snmp` commands to show that the module is functioning as expected

The `demo_module_2` is set up to allow you to generate code templates `me1FileTable.c` and `me1FileTable.h`. You can then compare the generated files to the files `demo_module_2.c` and `demo_module_2.h`.

The `mib2c` utility generates `me1FileTable.c`, which contains the `init_me1FileTable()` function. You should compare this function to the `init_demo_module_2()` function in the `demo_module_2.c` file.

Modifications for Simple Table Data Retrieval

In `demo_module_2.c`, the `init_demo_module_2` routine calls the `initialize_table_me1FileTable()` function. The `initialize_table_me1FileTable()` function registers the OID for the table handled by the function. The function also calls some Net-SNMP functions to initialize the tables.

You should provide the table data in this `initialize_table_me1FileTable()` function if needed. The `initialize_table_me1FileTable()` function performs the following:

Initialization

The `initialize_table_me1FileTable()` function performs the real table initialization, by performing tasks such as setting the maximum number of rows and columns.

OID Table Definition

The `initialize_table_me1FileTable()` function defines the table OID:

```
static oid me1FileTable_oid[] =
  {1,3,6,1,4,1,42,2,2,4,4,1,2,1};
```

Table Definition

The `initialize_table_me1FileTable()` function sets up the table's definition. This function specifies another function to call, `me1FileTable_get_first_data_point()`, to process the first row of data in the table. The function `me1FileTable_get_next_data_point()` is called to process the remaining rows in the table.

```
netsnmp_table_helper_add_indexes(table_info,
ASN_UNSIGNED, /* index: me1FileIndex */
0);

     table_info->min_column = 1;
     table_info->max_column = 4;

  /* iterator access routines */
iinfo->get_first_data_point =
           me1FileTable_get_first_data_point;
iinfo->get_next_data_point =
           me1FileTable_get_next_data_point;
iinfo->table_reginfo =
           table_info;
```

`iinfo` is a pointer to a `netsnmp_iterator_info` structure.

Master Agent Registration

The `initialize_table_me1FileTable()` function registers the table with the master agent:

```
netsnmp_register_table_iterator(my_handler, iinfo);
```

The table iterator is a helper function that modules can use to index through rows in a table. Functionally, the table iterator is a specialized version of the more generic table helper. The table iterator eases the burden of GETNEXT processing. The table iterator loops through all the data indexes retrieved through those function calls that should be supplied by the module that requests help. See the API documentation at `/usr/sfw/doc/sma_snmp/html/group__table__iterator.html` for more information on table iterator APIs.

Note that the input MIB file contains the specification of table and scalar data. However, when you run `mib2c` with `mib2c.iterate.conf` and specify the table node name, only template code for the simple table in the MIB is generated.

Data Retrieval From Large Simple Tables

Data retrieval from a simple table requires you to use the single, integer index subidentifier to index into an existing data structure.

With some modules, this underlying table might be relatively large, or only accessible through a cumbersome interface. Data retrieval might be very slow, particularly if performing a walk of a MIB tree requires the table to be reloaded for each variable requested. In these circumstances, a useful technique is to cache the table on the first read and use that cache for subsequent requests.

To cache the table, you must have a separate routine to read in the table. The routine uses two static variables. One variable is a structure or array for the data. The other variable is an additional timestamp to indicate when the table was last loaded. When a call is made to the routine to read the table, the routine can first determine whether the cached table is sufficiently new. If the data is recent enough, the routine can return immediately. The system then uses the cached data. If the cached version is old enough to be considered out of date, the routine can retrieve the current data. The routine updates the cached version of the data and the timestamp value. This approach is particularly useful if the data is relatively static, such as a list of mounted file systems.

Multiple SET Processing in `demo_module_2`

The `demo_module_2` example code shows how to perform a multiple OID set action. In this case, a file name and row status are provided.

When the agent processes a SET request, a series of calls to the MIB module code are made. These calls ensure that all SET requests in the incoming packet can be processed successfully. This processing allows modules the chance to get out of the transaction sequence early. If the module gets out of one transaction early, none of the transactions in the set are completed, in order to maintain continuity. However, this behavior makes the code for processing SET requests more complex. The following diagram is a simple state diagram that shows each step of the master agent's SET processing.

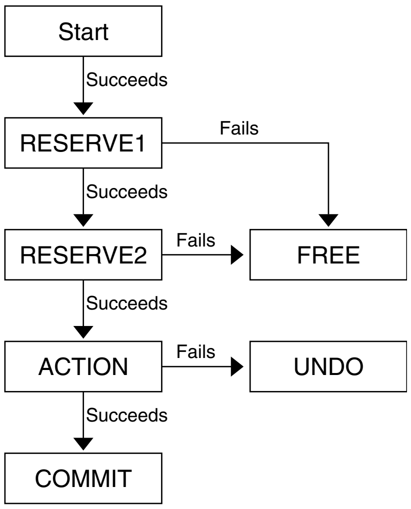

FIGURE 3–1 Set Processing State Diagram

An operation with no failures is illustrated by the vertical path on the left, in the preceding figure. If any of the MIB modules that are being acted upon returns an error, the agent branches to one of the failure states. The failure states are on the right side in the figure. These failure states require you to clean up and, where necessary, undo the actions of previous steps in your module.

See the `me1FileTable_handler()` function in the `demo_module_2` example code, for how to perform SET requests in different states. The following is list describes each of the states:

`case MODE_SET_RESERVE1`	Checks that the value being set is acceptable.
`case MODE_SET_RESERVE2`	Allocates any necessary resources. For example, calls to the `malloc()` function occur here.
`case MODE_SET_FREE`	Frees resources when one of the other values being SET failed for some reason.
`case MODE_SET_ACTION`	Sets the variable as requested and saves information that might be needed in order to reverse this SET later.

`case MODE_SET_COMMIT`	Operation is successful. Discards saved information and makes the change permanent. For example, writes to the `snmpd.conf` configuration file and frees any allocated resources.
`case MODE_SET_UNDO`	A failure occurred, so resets the variable to its previous value. Frees any allocated resources.

You can perform the set action using either of the following commands when you use the `demo_module_2` example:

```
snmpset -v1 -c private localhost me1FileTable.1.2.3 s "test"
```

```
snmpset -v1 -c private localhost .1.3.6.1.4.1.42.2.2.4.4.1.2.1.1.2.2 s "test"
```

These commands change the file that you want to monitor.

Note – In order to use the `snmpset` command to specify a different file name, you must have a private community string in the `snmpd.conf` file, which is located in `/etc/sma/snmp` or `$HOME/.snmp`.

General Tables

A general table differs from a simple table in at least one of the following ways:

- The table is not indexed with a single integer.

  For example, if the index is an IP address, the table is a general table.
- The maximum index cannot be determined easily.

  For example, the network interfaces table is a general table because it does not have a maximum index that you can determine.
- At any given point, some indexes might be invalid.

  For example, a table of currently running software might contain a row for a program that has just ended, but the table has yet to be updated. The table must be processed as a general table.
- The table data is not directly accessible.

  For example, the network interfaces table is maintained in the kernel and cannot be accessed directly.

The command that you use to generate code templates for general tables is the same command used for simple tables:

`mib2c -c mib2c.iterate.conf` *mibnode1 mibnode2 ...*

The `demo_module_3` code example shows how modify the templates appropriately to retrieve data from general tables.

`demo_module_3` Code Example for General Tables

The `demo_module_3` code example is provided to help you understand how to modify the code generated by the `mib2c` command to perform a data retrieval in a general table. The table example provides information for monitoring a list of files. The `demo_module_3` code example is located by default in the directory `/usr/demo/sma_snmp/demo_module_3`.

The `README_demo_module_3` file contains instructions that describe how to perform the following tasks:

- Generate code templates from a MIB that contains general table
- Compile source files to generate a shared library object that implements a module
- Set up the agent to dynamically load the module
- Test the module with `snmp` commands to show that the module is functioning as expected

The `demo_module_3` is set up to allow you to generate code templates `me1ContactInfoTable.c` and `me1ContactInfoTable.h`. You can then compare the generated files to the files `demo_module_3.c` and `demo_module_3.h`.

The `me1ContactInfoTable.c` and `me1ContactInfoTable.h` have been modified appropriately to retrieve data from general tables. You can use these files as a model for learning how to work with general tables in your own module. The instructions then explain how to compile the modified source files to create a functioning module.

The `demo_module_3` code was generated by using `mib2c` with the `-c mib2c.iterate.conf` option. Some functions have been added to implement a link list to provide the test data.

The example uses some dummy data to perform data retrieval for a two-index table. The code is similar to the `demo_module_2.c` with one extra index. The following code sets up the table with two indexes:

```
netsnmp_table_helper_add_indexes(table_info,
ASN_INTEGER, /* index: me1FloorNumber */
ASN_INTEGER, /* index: me1RoomNumber */
0);
```

Use care in returning the "NEXT" data when function `me1ContactInfoTable_get_next_data_point()` is called. For instance, the data in this table is presorted so the next data is conveniently pointed by the `pNext` pointer in this example code:

```
me1ContactEntry* nextNode = (me1ContactEntry*) *my_loop_context;
          nextNode = nextNode->pNext;
```

If your implementation is more complicated, make sure the OIDs are increased incrementally, (*xxx*.1.1, *xxx*.1.2,).

The input MIB file contains the specification of tables and scalars. When you run `mib2c -c mib2c.iterate.conf` on a general table node, template code is generated only for the general table in the MIB.

CHAPTER 4

Storing Module Data

This chapter discusses how a module can store data that persists when the agent is restarted.

The chapter includes the following topics:

About Storing Module Data

You might want your module to store *persistent data*. Persistent data is information such as configuration settings that the module stores in a file and reads from that file. The data is preserved across restarts of the agent.

Modules can store tokens with assigned values in module-specific configuration files. A configuration file is created manually. Tokens can be written to the file or read from the file by a module. The module registers handlers that are associated with the module's specific configuration tokens.

Configuration Files

The `snmp_config`(4) man page discusses SNMP configuration files in general. The man page documents the locations where the files can be stored so the agent can find the files. These locations are on the default search path for SNMP configuration files.

For your modules, the best location to store configuration files is in a `$HOME/.snmp` directory, which is on the default search path. You can also set the `SNMPCONFPATH` environment variable if you want to use a non-default location for configuration files.

When you create your own configuration file, you must name the file *module*`.conf` or *module*`.local.conf`. You must place the file in one of the directories on the SNMP configuration file search path.

Note – You might find that the Net-SNMP routines write your module's configuration file to the `/var/sma_snmp` directory. The routines make updates to that version of the file. However, the routines can find the configuration file in other locations when the module needs to initially read the file.

Defining New Configuration Tokens

Configuration tokens are used by modules to get persistent data during runtime. When your module uses custom configuration tokens, you should create one or more custom configuration files for the module. You might also choose to create one configuration file for several related modules. You can define new tokens in the custom configuration file.

Custom tokens must use the same format as the directives in `snmpd.conf`. One token is defined in each line of the configuration file. The configuration tokens are written in the form:

Token Value

For example, your token might be:

```
my_token 4
```

Modules should not define custom tokens in the SNMP configuration file, `/etc/sma/snmp/snmpd.conf`. If a module stores tokens in `/etc/sma/snmp/snmpd.conf`, namespace collisions can potentially occur. See "Avoiding Namespace Collisions" on page 36 for more information about namespace collisions.

Implementing Persistent Data in a Module

The module can register handlers that are associated with tokens in a module-specific configuration file with the `register_config_handler()` function. The handlers can then be used later in the module for a specific task.

The `register_config_handler()` is defined as follows:

```
register_config_handler (const char *type_param, const char *token,
                    void(*parser)(const char *, char *),
                    void(*releaser)(void), const char *help)
```

The first argument to this function designates the base name of the configuration file, which should be the same as the name of the module. For example, if the first argument is `my_custom_module`, then the agent infrastructure looks for the configuration tokens in the file `my_custom_module.conf`. Note that you must create the configuration file manually before the module can use the file.

The second argument to this function designates the configuration token that the module is looking for.

For more information about `register_config_handler()` and other related functions, see the API documentation in `/usr/sfw/doc/sma_snmp/html/group__read__config.html`. You can also look at `/usr/demo/sma_snmp/demo_module_5/demo_module_5.c` to see how the function is used.

Storing Persistent Data

Your module must use the `read_config_store_data()` and `read_config_store()` functions together with callback functions to store data.

Your module must first register a callback with the `snmp_register_callback()` function so that data is written to the configuration file when the agent shuts down.

The `snmp_register_callback()` function is as follows:

```
int snmp_register_callback(int major,
                           int minor,
                           SNMPCallback *new_callback,
                           void *arg);
```

You must set *major* to `SNMP_CALLBACK_LIBRARY`, set *minor* to `SNMP_CALLBACK_STORE_DATA`. When *arg* is not set to `NULL`, arg is a void pointer used whenever the *new_callback* function is exercised.

The prototype to your callback function, the *new_callback* pointer, is as follows:

```
int (SNMPCallback) (int majorID,
                    int minorID,
                    void *serverarg,
                    void *clientarg);
```

See the API documentation for more information about setting up callback registrations with the agent at `/usr/sfw/doc/sma_snmp/html/group__callback.html`.

The `read_config_store_data()` function should be used to create the token-value pair that is to be written into the module's configuration file. The `read_config_store()` function actually does the storing when the registered callbacks are exercised upon agent shutdown.

Note – When your module stores persistent data, you might find that the configuration file is written to the `/var/sma_snmp` directory. Modified token-value pairs are appended to the file, rather than overwriting the previous token-value pairs in the file. The last values that were defined in the file are the values that are used.

Reading Persistent Data

Data is read from a module's configuration file into the module by using the `register_config_handler()` function. For example, you can call the function as follows:

```
register_config_handler("my_module", "some_token",
                            load_my_tokens, NULL, NULL);
```

Whenever the token `some_token` is read by the agent in `my_module.conf` file, the `load_my_tokens()` function is called with token name and value as arguments. The `load_my_tokens()` function is invoked. The data can be parsed by using the `read_config_read_data()` function.

`demo_module_5` Code Example for Persistent Data

The `demo_module_5` code example demonstrates the persistence of data across agent restart. The demo is located in the directory `/usr/demo/sma_snmp/demo_module_5` by default.

This module implements `SDK-DEMO5-MIB.txt`. The `demo_module_5.c` and `demo_module_5.h` templates were renamed from the original templates `me5FileGroup.c` and `me5FileGroup.h` that were generated with the `mib2c` command. The name of the initialization function is changed from `init_me5FileGroup` to `init_demo_module_5`.

See the `README_demo_module_5` file in the `demo_module_5` directory for the procedures to build and run the demo.

Storing Persistent Data in `demo_module_5`

This example stores configuration data in the `/var/sma_snmp/demo_module_5.conf` file.

In `demo_module_5.c`, the following statement registers the callback function. The callback function is called whenever the agent sees that module data needs to be stored, such as during normal termination of the agent.

```
snmp_register_callback(SNMP_CALLBACK_LIBRARY,
                                SNMP_CALLBACK_STORE_DATA,
                                demo5_persist_data,
                                NULL);
```

The `demo5_persist_data()` function uses `read_store_config` to store data:

```
int demo5_persist_data(int a, int b, void *c, void *d)
{
    char            filebuf[300];

    sprintf(filebuf, "demo5_file1 %s", file1);
    read_config_store(DEMO5_CONF_FILE, filebuf);

    sprintf(filebuf, "demo5_file2 %s", file2);
    read_config_store(DEMO5_CONF_FILE, filebuf);

    sprintf(filebuf, "demo5_file3 %s", file3);
    read_config_store(DEMO5_CONF_FILE, filebuf);

    sprintf(filebuf, "demo5_file4 %s", file4);
    read_config_store(DEMO5_CONF_FILE, filebuf);

}
```

In `demo_module_5`, a new file can be added for monitoring, by using the `snmpset` command. The commit phase of the `snmpset` request uses the `read_config_store ()` function to store file information:

```
case MODE_SET_COMMIT:
    /*
     * Everything worked, so we can discard any saved information,
     * and make the change permanent (e.g. write to the config file).
     * We also free any allocated resources.
     *
     */Persist the file information */

    snprintf(&filebuf[0], MAXNAMELEN, "demo5_file%d %s",
            data->findex, data->fileName);



        read_config_store(DEMO5_CONF_FILE, &filebuf[0]);

        /*
        * The netsnmp_free_list_data should take care of the allocated
        * resources
        */
```

The persistent data is stored in the `/var/sma_snmp/demo_module_5.conf` file.

Reading Persistent Data in `demo_module_5`

Data is read from the configuration files into a module by registering a callback function to be called whenever an relevant token is encountered. For example, you can call the function as follows:

```
register_config_handler(DEMO5_CONF_FILE, "demo5_file1",
demo5_load_tokens, NULL, NULL);
```

Whenever the `demo5_file1` token in the `demo_module_5.conf` file is read by the agent, the function `demo5_load_tokens()` is called with token name and value as arguments. The `demo5_load_tokens()` function stores the token value in appropriate variables:

```
void
demo5_load_tokens(const char *token, char *cptr)
{

    if (strcmp(token, "demo5_file1") == 0) {
        strcpy(file1, cptr);
    } else if (strcmp(token, "demo5_file2") == 0) {
        strcpy(file2, cptr);
    } else if (strcmp(token, "demo5_file3") == 0) {
        strcpy(file3, cptr);
    } else if (strcmp(token, "demo5_file4") == 0) {
        strcpy(file4, cptr);
    } else {
        /* Do Nothing */
    }

    return;

}
```

Using `SNMP_CALLBACK_POST_READ_CONFIG` in `demo_module_5`

A few seconds elapse after agent startup while all configuration tokens are read by the module. During this interval, the module should not perform certain functions. For example, until the persistent file names are read from `/var/sma_snmp/demo_module_5.conf` into the module, the file table cannot be populated. To handle these cases, a callback function can be set. This callback function is called when the process of reading the configuration files is complete. For example, you might call the function as follows:

```
snmp_register_callback(SNMP_CALLBACK_LIBRARY,
 SNMP_CALLBACK_POST_READ_CONFIG, demo_5_post_read_config, NULL);
```

The `demo_5_post_read_config()` function is called after the configuration files are read. In this example, the `demo_5_post_read_config()` function populates the file table, then registers the callback function for data persistence.

```
int
demo5_post_read_config(int a, int b, void *c, void *d)
{   if (!AddItem(file1))
     snmp_log(LOG_ERR, "Failed to add instance in init_demo_module_5\n");
  if (!AddItem(file2))
     snmp_log(LOG_ERR, "Failed to add instance in init_demo_module_5\n");
  if (!AddItem(file3))
     snmp_log(LOG_ERR, "Failed to add instance in init_demo_module_5\n");
  if (!AddItem(file4))
     snmp_log(LOG_ERR, "Failed to add instance in init_demo_module_5\n");


snmp_register_callback
(SNMP_CALLBACK_LIBRARY, SNMP_CALLBACK_STORE_DATA,
demo5_persist_data, NULL);

}
```

CHAPTER 5

Implementing Alarms

This chapter explains how to implement alarms in modules. The `demo_module_4` is used to illustrate techniques.

The chapter contains the following topics:

Refresh Intervals

Refresh intervals, also known as automatic refresh, can be implemented in the System Management Agent. You can use a callback mechanism that calls a specified function at regular intervals. Data refresh can be implemented by the `snmp_alarm_register ()` function. In `demo_module_4`, the load data is refreshed at a configurable time interval, 60 seconds in this example, using the following callback:

```
snmp_alarm_register(60, SA_REPEAT, refreshLoadAvg, NULL);

void refreshLoadAvg(unsigned int clientreg, void *clientarg){

    // Refresh the load data here

}
```

The `snmp_alarm_register()` function can be included in the `init_()`*module*() function so that the refresh interval is set during the initialization of the module.

Asynchronous Trap Notification

Typically, checking for trap conditions is done in the following sequence:

1. Get current data for a particular node.
2. Compare the data with a threshold to check if the trap condition is met.
3. Send a trap to the manager if the condition is met.

Steps 2 and 3 are implemented in SMA by calling an algorithm after data for a node is acquired. The algorithm determines if an alarm condition is met. The algorithm in most cases involves comparing the current data with the threshold. If the algorithm indicates that an alarm condition is met, the appropriate trap functions are called to issue a trap. In `demo_module_4`, steps 2 and 3 are performed in the following function:

```
void refreshLoadAvg(unsigned int clientreg, void *clientarg) {

    // Refresh Load data

    // Check if Load data crossed thresholds, send trap if necessary.
    check_loadavg1_state();
    check_loadavg5_state();
    check_loadavg15_state();

}
```

The `check_loadavg_state` functions compare the current load data with thresholds. The functions also send the traps if necessary.

The module must use a trap function such as `send_v2trap()` to send a trap to the manager. For more information on SNMP trap APIs, see `/usr/sfw/doc/sma_snmp/html/group__agent__trap.html`. The SNMP trap notifications are defined in `SNMP-NOTIFICATION-MIB.txt`. For `demo_module_4`, the trap notifications are defined in `SDK-DEMO4-MIB.txt`.

Thresholds for Sending Traps

In the System Management Agent, any configurable data can be stored in a module-specific configuration file. Data from this file can be loaded into the module at the time of module initialization. Data is read from the configuration files into a module by registering a callback function to be called whenever an interesting token is encountered.

```
register_config_handler("demo_module_4", "threshold_loadavg1",
read_load_thresholds, NULL, NULL);
```

In this example demo_module_4, whenever a threshold_loadavg1 token is read by the agent in the demo_module_4.conf file, the read_load_thresholds() function is called, with token name and value as arguments. The read_load_thresholds() function stores the token value in appropriate variables and uses these thresholds to determine alarm conditions. For more information on the register_config_handler APIs, see the documentation in /usr/sfw/doc/sma_snmp/html/group__read__config.html.

demo_module_4 Code Example for Alarms

The demo_module_4 code example is provided to help you understand how to implement alarms. The demo is by default located in the directory /usr/demo/sma_snmp/demo_module_4. The README_demo_module_4 file in that directory contains instructions that describe how to do the following tasks:

- Compile source files to generate a shared library object that implements a module
- Set up the agent to dynamically load the module
- Test the module with snmp commands to show that the module is functioning as expected

The demo_module_4 module implements SDK-DEMO4-MIB.txt. The me4LoadGroup.c and me4LoadGroup.h files were generated with the mib2c command and then modified.

Module data is maintained in the following variables:

loadavg1	Stores data for me4SystemLoadAvg1min
loadavg5	Stores data for me4SystemLoadAvg5min
loadavg15	Stores data for me4SystemLoadAvg15min

The demo_module_4 module refreshes data every 60 seconds. During refresh intervals, the module also checks whether trap conditions are met. If trap conditions are met, an SNMPv2 trap is generated by the module. The trap condition in this module is a simple comparison of current data with a threshold value. If the threshold is crossed, a trap is generated. The threshold data can be configured through the file demo_module_4.conf, which is installed in $HOME/.snmp.

When an snmpget request for these variables arrives, the following functions are called:

- int get_me4SystemLoadAvg1min()
- int get_me4SystemLoadAvg5min()

- `int get_me4SystemLoadAvg15min()`

These accessory functions refresh the load data by calling the `refreshLoadAvg()` function to return the current load value. However, the reload occurs only in response to a GET request. The load data must also be refreshed asynchronously without waiting for GET requests from the manager. Asynchronous refreshing allows trap conditions to be checked continuously in order to alert the manager of any problems. You can refresh the load data without a request from the manager by registering a callback function to be called at regular intervals. For example, you can call the function as follows:

```
snmp_alarm_register(60, SA_REPEAT, refreshLoadAvg, NULL)
```

This function causes the `refreshLoadAvg()` function to be called every 60 seconds. You can enable a manager to configure this interval by introducing a token to represent this value in the `demo_module_4.conf` file.

See the API documentation at `/usr/sfw/doc/sma_snmp/html /group__snmp__alarm.html` for more information on `snmp_alarm_register()` functions.

Reading Data From the `demo_module_4.conf` Configuration File

Data is read from the configuration files into a module by registering a callback function to be called whenever an appropriate token is encountered. For example, you can call the function as follows:

```
register_config_handler(demo_module_4, threshold_loadavg1,
read_load_thresholds, NULL, NULL);
```

Whenever a `threshold_loadavg1` token in the `demo_module_4` file is read by the agent, the function `read_load_thresholds()` is called with token name and value as arguments. The `read_load_thresholds()` function stores the token value in appropriate variables:

```
void
read_load_thresholds(const char *token, char *cptr)
{

    if (strcmp(token, "threshold_loadavg1") == 0) {
        threshold_loadavg1=atof(cptr);
    } else if (strcmp(token,"threshold_loadavg5") == 0) {
        threshold_loadavg5=atof(cptr);
    } else if (strcmp(token,"threshold_loadavg15") == 0) {
        threshold_loadavg15=atof(cptr);
    } else {
        /* Do nothing */
    }
```

```
        return;

}
```

See the API documentation about `register_config_handler()` in `/usr/sfw/doc/sma_snmp/html/group__read__config.html` for more information.

Using `SNMP_CALLBACK_POST_READ_CONFIG` in `demo_module_4`

A few seconds elapse after agent startup while all configuration tokens are read by the module. During this interval, the module should not perform certain functions. For example, until the threshold settings are read from configuration files into the module, trap condition checks should not be performed. To handle these cases, a callback function can be set. This callback function is called when the process of reading the configuration files is complete. For example, you can call the function as follows:

```
snmp_register_callback(SNMP_CALLBACK_LIBRARY,
SNMP_CALLBACK_POST_READ_CONFIG,demo_4_post_read_config, NULL);
```

The `demo_4_post_read_config()` function is called after the configuration files are read. In this example, the `demo_4_post_read_config()` function registers refresh callbacks:

```
int demo_4_post_read_config(int a, int b, void *c, void *d)
{

    /* Refresh the load data every 60 seconds */
    snmp_alarm_register(60, SA_REPEAT, refreshLoadAvg, NULL);

    /* Acquire the data first time */
    refreshLoadAvg(0,NULL);

}
```

Generating Traps in `demo_module_4`

The `refreshLoadAvg()` function is called at regular intervals to refresh data. Immediately after data is refreshed, the `refreshLoadAvg()` function checks for trap conditions by calling the following functions:

- `check_loadavg1_state()`
- `check_loadavg5_state()`
- `check_loadavg15_state()`

In me4LoadGroup.c, a module property could be in one of two states: OK or ERROR. When the current data value crosses the threshold, the state is set to ERROR. A trap is then generated. The check functions have the following algorithm:

```
check_loadavg1_state() {

    // Step-1: check condition
    if (currentData > threshold_loadavg1) new_loadavg1_state = ERROR;

    // Step-2: Generate trap if necessary
        if (new_loadavg1_state > prev_loadavg1_state) {
                // Send trap
                prev_loadavg1_state=new_loadavg1_state;
        } else if(new_loadavg1_state == prev_loadavg1_state) {
                /* No Change in state .. Do nothing */
        } else if (new_loadavg1_state < prev_loadavg1_state) {
                if (new_loadavg1_state == OK) {
                        prev_loadavg1_state=OK;
            // Send OK trap
                }
        }
}
```

When the check indicates that the threshold has been crossed, the send_v2trap function is used to generate an SNMPv2 trap. The trap OID and the varbinds are as specified in the SDK-DEMO4-MIB.txt MIB. For more information on SNMP trap APIs, see /usr/sfw/doc/sma_snmp/html/group__agent__trap.html.

CHAPTER 6

Deploying Modules

This chapter discusses the ways to deploy your module. The chapter provides information to help you decide whether you should use a subagent or a dynamically loaded module. Examples of deploying demonstration modules as subagents and dynamically loaded modules are included.

This chapter contains the following topics:

Overview of Module Deployment

With the System Management Agent, you have the following choices for deploying a module:

- Load the module dynamically.

  When you load a module dynamically, the module is included within the SNMP agent without the need to recompile and relink the agent binary. This method is the only supported way to load a module into the System Management Agent. You cannot recompile the agent.

  Details of the module to load are specified in the configuration file. At runtime, the agent reads the configuration file. The agent locates the module files that are listed in the configuration file. The agent then merges the modules into the agent process image.

- Implement the module as an AgentX subagent.

When you use a subagent, the module is embedded in an external application. The external application contains code to set up the application to run as an AgentX subagent. The SNMP agent's configuration file specifies that the agent is the AgentX master agent. When the external application starts, the module's OIDs are registered with the SNMP agent. The subagent application and the agent use the AgentX protocol to communicate.

These deployment methods have advantages and disadvantages, which are discussed in "Choosing Dynamic Modules or Subagents" on page 66. However, the way that you develop your module and the content of your module have no bearing on how you deploy your module. You can use the same module, without modification, with either deployment method.

Choosing Dynamic Modules or Subagents

In general, when you are first developing and testing your module, you should dynamically load the module in the master agent. This method reduces the complexity while you work out any problems in the module. When you are ready to deploy a module, you should compile the module in a subagent instead of dynamically loading into the master agent. By using subagents, you can more easily isolate problems in the module.

However, sometimes a subagent is not the optimal deployment method. Use the following criteria to determine when to load a module into a master agent instead of a subagent:

- If more than five requests per second are targeted to the module's MIB, you should consider loading the module into the master agent.
- If your module queries the SYSTEM group often, and queries the IP branch very rarely, load the module into the master agent.
- If your module queries the IP branch of the MIB very often, load the module in the subagent. The IP group of the module MIB is six times more computationally costly compared to the SYSTEM group.

The following table summarizes the primary advantages and disadvantages of dynamically loaded modules and subagents.

TABLE 6–1 Advantages and Disadvantages of Deployment Methods

Deployment Method	Advantages	Disadvantages
Dynamically loaded	Less complexity compared to subagent approach	Incurs a slightly greater load on agent at startup.
		Makes the master agent more vulnerable, especially if the module has quality and performance problems. For example, a module with quality problems might have a memory violation, which can crash the master agent. A module with performance problems might consume too many system resources, such as CPU time and memory. These problems might overload the master agent, causing the master agent not to function properly in processing other requests.
AgentX subagent	Isolates the module processing from the agent	Incurs an extra cost to the master agent by causing the agent to build packets to transport requests between the master agent and the subagent. The master agent performs an extra step of both encoding and decoding for every incoming request that is targeted to the subagent. If the subagent gets too many requests, the time spent on additional encoding and decoding might be excessive.

Loading Modules Dynamically

The simplest way to load modules dynamically is to restart the agent after you add entries to the configuration file. Dynamic loading is the best method to use while you are developing and testing a module. Most of the demonstration modules in `/usr/demo/sma_snmp` use dynamic loading. You should use the procedure "How to Dynamically Load a Module and Restart the Agent" on page 68 during the development and testing phase.

When you are using the module in a production environment, that environment might require you not to restart the agent. If you want to load modules without restarting the agent, you should use the procedure "How to Dynamically Load a Module Without Restarting the Agent" on page 68.

▼ How to Dynamically Load a Module and Restart the Agent

Steps

1. **Copy the module shared library object to a `lib` directory.**

   You should keep your `.so` files in a directory that is writable by non-root users.

2. **As root, edit the agent's configuration file to enable the agent to dynamically load the module.**

   In the `/etc/sma/snmp/snmpd.conf` file, add a line that is similar to the following, where `testmodule` is the name of the module.

   ```
   dlmod testmodule /home/username/snmp/lib/testmodule.so
   ```

3. **As root, restart the `snmpd` agent by typing the following command.**

   ```
   # /etc/init.d/init.sma restart
   ```

   The module should now be loaded. You can use `snmpget` and `snmpset` commands to access the module's data to confirm that the module is loaded. You should make sure your MIB can be located by the `snmpget` and `snmpset` commands by setting your `MIBDIRS` and `MIBS` environment variables, as described in "Setting MIB Environment Variables" on page 32.

   Tip – To unload a module, you would remove the `dlmod` line from the `snmpd.conf` file and restart the agent.

▼ How to Dynamically Load a Module Without Restarting the Agent

The UCD-DLMOD-MIB provides MIB entries for the module name, path, and status. By setting these MIB entries, you can cause the agent to load or unload the module without restarting the agent.

Note – This procedure causes the module to be loaded only for the current session of the agent. If you want the module to be loaded each time the agent starts, you should add a `dlmod` line to the `snmpd.conf` file. The process of adding the line is described in Step 2 of the previous procedure. Do not restart the agent after adding the line.

Steps

1. **View the `UCD-DLMOD-MIB.txt` file in `/etc/sma/snmp/mibs`.**

   Look for the `DlmodEntry` and `dlmodStatus` entries, which appear as follows:

   ```
   DlmodEntry ::= SEQUENCE {
        dlmodIndex  Integer32,
   ```

```
     dlmodName   DisplayString,
     dlmodPath   DisplayString,
     dlmodError  DisplayString,
     dlmodStatus INTEGER
}

dlmodStatus OBJECT-TYPE
     SYNTAX      INTEGER {
                     loaded(1),
                     unloaded(2),
                     error(3),
                     load(4),
                     unload(5),
                     create(6),
                     delete(7)
                 }
     MAX-ACCESS  read-write
     STATUS      current
     DESCRIPTION
         "The current status of the loaded module."
     ::= { dlmodEntry 5 }
```

`DlmodEntry` defines a row in a table of dynamically loaded modules. A table row describes an instance by defining an index, name, path, error code, and status code. You need to set the name, path, and status of the first empty row of the table.

2. **Type the following command to check the first row of the table. The command can tell you whether an instance of a dynamically loaded module already exists in the table.**

```
$ /usr/sfw/bin/snmpget -v 1 -c public localhost UCD-DLMOD-MIB::dlmodStatus.1
Error in packet
Reason: (noSuchName) There is no such variable name in this MIB.
Failed object: UCD-DLMOD-MIB::dlmodStatus.1
```

This response indicates that no other dynamic module is defined as instance 1. If you get back a positive response, examine `dlmodStatus.2` with the same command.

3. **Create an instance for your module in the table by typing the following command:**

```
$ /usr/sfw/bin/snmpset -v 1 -c private localhost \
UCD-DLMOD-MIB::dlmodStatus.1 i create

UCD-DLMOD-MIB::dlmodStatus.1 = INTEGER: create(6)
```

4. **Repeat the `snmpget` command to show the status of the first instance.**

```
$ /usr/sfw/bin/snmpget -v 1 -c public localhost \
UCD-DLMOD-MIB::dlmodStatus.1
UCD-DLMOD-MIB::dlmodStatus.1 = INTEGER: unloaded(2)
```

The instance now exists, but the module is unloaded currently.

5. **Set the name and path to the module that you want to load. Type a command that is similar to the following:**

```
$ /usr/sfw/bin/snmpset -v 1 -c private localhost \
UCD-DLMOD-MIB::dlmodName.1 s "testmodule" \
UCD-DLMOD-MIB::dlmodPath.1 s "/home/username/lib/testmodule.so"
UCD-DLMOD-MIB::dlmodName.1 = STRING: testmodule
UCD-DLMOD-MIB::dlmodPath.1 = STRING: /home/username/lib/testmodule.so
```

`testmodule` is the name of your module.

6. **Load the module by typing the following command:**

```
$ /usr/sfw/bin/snmpset -v 1 -c private localhost \
UCD-DLMOD-MIB::dlmodStatus.1 i load
UCD-DLMOD-MIB::dlmodStatus.1 = INTEGER: load(4)
```

   This command sets the `dlmodStatus.1` variable to `load`.

7. **Confirm that the module was loaded by typing the following command:**

```
$ /usr/sfw/bin/snmpget  -v 1 -c public localhost \
UCD-DLMOD-MIB::dlmodStatus.1
UCD-DLMOD-MIB::dlmodStatus.1 = INTEGER: loaded(1)
```

   The response indicates that the module is loaded.

8. **(Optional) Unload the module by typing the following command:**

```
$ /usr/sfw/bin/snmpset -v 1 -c private localhost \
UCD-DLMOD-MIB::dlmodStatus.1 i unload
UCD-DLMOD-MIB::dlmodStatus.1 = INTEGER: unload(5)
```

9. **(Optional) Confirm that the module was unloaded by typing the following command:**

```
$ /usr/sfw/bin/snmpget -v 1 -c public localhost \
UCD-DLMOD-MIB::dlmodStatus.1
Timeout: No Response from localhost.
```

   The lack of response from `localhost` indicates that the module is unloaded.

Using Subagents

Using subagents with an extensible SNMP agent avoids the problem of having one very large SNMP agent. Before subagents were devised, an SNMP agent had to be recompiled to add new management objects in MIBs. The subagent approach provides the ability to dynamically add management objects to an agent without recompiling the agent. The need to standardize the way in which agents and subagents work together led to the development of the AgentX protocol.

AgentX Protocol

The AgentX protocol enables subagents to connect to the master agent. The protocol also enables the master agent to distribute received SNMP protocol messages to the subagents.

The AgentX protocol defines an SNMP agent to consist of one master agent entity and other subagent entities. The master agent runs on the SNMP port, and sends and receives SNMP messages as specified by the SNMP framework documents. The master agent does not access the subagents' management information directly. The subagents do not handle SNMP messages, but subagents do access their management information. In short, the master agent handles SNMP for the subagents, and *only* handles SNMP. The subagent handles manipulation of management data but does not handle SNMP messages. The responsibilities of each type of agent are strictly defined. The master agent and subagents communicate through AgentX protocol messages. AgentX is described in detail by RFC 2741. See `http://www.ietf.org/rfc/rfc2741.txt`

The SMA performs in the role of the master agent. Subagents that you create can add management objects to the agent.

Functions of a Subagent

An AgentX subagent performs the following functions:

- Initiates AgentX sessions with the master agent
- Registers MIB regions with the master agent
- Instantiates managed objects
- Binds object IDs (OIDs) within its registered MIB regions to actual variables
- Performs management operations on variables
- Initiates notifications, or traps

Deploying a Module as a Subagent

You can embed a MIB module that was written for the SMA into an external application. This application can be run either as an SNMP master agent or an AgentX subagent. Generally, you should run the SMA as the master agent, and set up your application as a subagent. The subagent attaches to the master agent, and registers its MIB with the master agent. By running the SMA as the master agent, you can easily add and remove subagents while the master agent continues to run. In this way, the agent can continue to communicate with network management applications.

SMA provides Net-SNMP API functions that enable you to embed an SNMP agent or AgentX subagent into an external application. In your application code, you must initialize your module, the SNMP library, and the SNMP agent library. This initialization is done slightly differently depending on whether the application is to run as a master agent or an AgentX subagent.

The functions that you use in the agent application include:

- `init_agent(char *name)`

  Initializes the embedded agent. This function must be called before the `init_snmp()` call. The `name` is used to specify what configuration file to read when `init_snmp()` is called later. See the API documentation at `/usr/sfw/doc/sma_snmp/html/group__library.html` for more information.
- `init_`*module*()

  Initializes your module. This function must be called after the agent is initialized.
- `init_snmp(char *name)`

  Initializes the SNMP library, which causes the agent to read the application's configuration file. The configuration file can be used to configure access control, for instance. See the `snmp_config`(4) and `snmpd.conf`(4) man pages for more information about configuration files.
- `snmp_shutdown(char *name)`

  Shuts down the subagent, saving any needed persistent data. See the API documentation at `/usr/sfw/doc/sma_snmp/html/group__library.html` for more information.

You must also link against the Net-SNMP libraries in your subagent application. The command

```
net-snmp-config --agent-libs
```

displays a list of libraries you need.

The `demo_module_8` code example shows you how to create a subagent that calls a module that returns load averages.

`demo_module_8` Code Example for Implementing a Subagent

The `demo_module_8` code example demonstrates how to deploy a module in a subagent. The demo is by default located in the directory `/usr/demo/sma_snmp/demo_module_8`. The `README_demo_module_8` file within that directory includes procedures for building and running the sample module and subagent program.

Subagent Security Guidelines

You must be aware of the following security considerations in writing subagents that use the AgentX protocol:

- The AgentX protocol does not contain a mechanism for authorizing or refusing to initiate sessions. Access control between subagents and master agent must be done at a lower layer, such as the transport layer.

  The SMA supports only UNIX domain sockets for communication between the master agent and subagents. As a result, the master agent and subagents must run on the same host.

  In open source Net-SNMP, the master agent and subagent can be on different hosts. The agents must then use UDP and TCP ports for the AgentX communication. Currently, the AgentX protocol provides no inherent security when using UDP and TCP ports. To reduce security risks, the SMA does not allow subagents to use UDP and TCP ports.
- The AgentX protocol does not define any access control mechanism. The protocol also does not contain a mechanism for authorizing or refusing sessions.
- A subagent can register any subtree. Potentially, a malicious subagent could register an unauthorized subtree of sensitive information. That subagent could then see modification requests to those objects in the tree. A malicious subagent might also give answers to SNMP manager queries. These answers might cause the manager to perform an action that leads to information disclosure or other damage.

CHAPTER 7

Multiple Instance Modules

This chapter describes how to implement a module to allow more than one instance of the module to run on a host. The chapter also describes how to dynamically update modules with multiple instances.

The following topics are discussed:

Implementing Multiple Instances of a Module

For some types of modules, multiple instances of the module can be run simultaneously on a single host. For example, consider a module that monitors the status of a single printer. For a system with several printers, the printer-monitoring module must be loaded multiple times, once for each printer. In that scenario, several separate instances of the printer module must be running simultaneously. For such modules, you must distinguish the different instances that are loaded and running.

SNMPv2 introduced the concept of *contexts* to identify MIB modules that can have multiple instances. Each SNMP context is represented by a separate MIB subtree.

In SMA, you can implement multiple instances of a module only when the agent is configured to use SNMPv3. You need to specify an SNMPv3 user and password when loading and unloading modules. You specify an instance name by assigning a string to the `contextName` member of the `netsnmp_handler_registration` struct in the module.

The following procedure tells you how to implement multiple-instance modules. The procedure uses examples from `demo_module_6`, which you can adapt to your own module.

▼ How To Implement Multiple Instance Modules

Steps

1. **As root, stop the agent if the agent is already running.**

   ```
   # /etc/init.d/init.sma stop
   ```

2. **Set up an SNMPv3 user.**

   For example, set up user name `myuser` with password `mypassword` as follows:

   ```
   # $ /usr/sfw/bin/net-snmp-config
   --create-snmpv3-user myuser

   Enter authentication pass-phrase:
   mypassword

   Enter encryption pass-phrase:
   [press return to reuse the authentication pass-phrase]
   ```

3. **Edit the module to register context names that the module handles.**

   Find the `init_`*module* routine in the module. Add code to register context names that the module handles.

   For example, you might add the following code:

   ```
   void
   init_filesize(void)
   {
   // Declare the OID
   static oid filesize_oid[] = { 1,3,6,1,4,1,42,2,2,4,4,6,1,1,0 };


   // Declare a registration handler
   netsnmp_handler_registration *myreg1;

   // Declare pointers to character arrays initialized
   // to the context name strings
   char *filexcon = "fileX";
   char *fileycon = "fileY";

   // Create a registration handler for the OID.
   // filesize is the name of handler.
   // get_filesize is the function to call when an SNMP
   // request for the OID is received, filesize_oid is the
   // OID for which the handler is registered,
   // OID_LENGTH(filesize_oid) passes the length of the
   // OID array to the agent.
   // HANDLER_CAN_RONLY is a constant that specifies that
   ```

```
// this handler only handles get requests.
myreg1 = netsnmp_create_handler_registration
("filesize", get_filesize,
             filesize_oid, OID_LENGTH(filesize_oid),
             HANDLER_CAN_RONLY);

// Assign the string fileX to the contextName member of the
// netsnmp_handler_registration struct
myreg1->contextName=filexcon;

// Register the netsnmp_handler_registration struct with the
// agent.  netsnmp_register_read_only is a helper function
// that notifies the agent that this module only handles snmp
// get requests.
netsnmp_register_read_only_instance(myreg1);
}
```

`demo_module_6` Code Example for Multiple Instance Modules

The `demo_module_6` code is located by default in `/usr/demo/sma_snmp/demo_module_6`. The `README_demo_module_6` file within that directory contains instructions that describe how to perform the following tasks:

- Compile source files to generate a shared library object that implements a module
- Set up the agent with an SNMPv3 user
- Set up the agent to dynamically load the module
- Test the module with `snmp` commands to show that the module is functioning as expected

The `demo_module_6` example shows how to write a module that registers an object in two different contexts. The example also shows how to check for the `contextName` in a request and return a different value depending on the value of the `contextName`.

`demo_module_6` registers one object, `filesize`, in two different contexts, `fileX`, and `fileY`. The OIDs are registered by using a read-only instance handler helper. The OIDs do not need to be read-only. You could also register the OIDs using any of the SMA instance handler helper APIs.

The function `get_filesize()` is registered to handle GET requests for instances of the `filesize` object. The `get_filesize()` function checks the `contextName` in the `reginfo` structure that is passed to the function by the SMA. If the value of `contextName` is `fileX`, the function returns `fileX_data`, which has been set to the integer 111. If the value of `contextName` is `fileY`, the function returns `fileY_data`, which has been set to the integer 999.

Enabling Dynamic Updates to a Multiple Instance Module

When you perform a dynamic update to a module, you use a command to modify a module that is loaded and running with System Management Agent. The SMA does not provide a mechanism for dynamically adding and removing instances of managed objects in a multi-instance module. However, you can code your module to enable an administrator or application to use the `snmpset` command to update the module.

The `demo_module_7` code example is used to show how to update a module that has been registered with the agent.

`demo_module_7` Code Example for Dynamic Updates of Multiple Instance Modules

The `demo_module_7` code example shows how to implement multiple instance modules. The demo is by default located in the directory `/usr/demo/sma_snmp/demo_module_7`. The `README_demo_module_7` file in that directory contains instructions that describe how to perform the following tasks:

- Compile source files to generate a shared library object that implements a module
- Set up the agent with an SNMPv3 user
- Set up the agent to dynamically load the module
- Test the module with `snmp` commands to show that the module is functioning as expected

Modifying the `demo_module_7` Code

The following procedure lists the steps you should follow to enable your module to be dynamically updated. The procedure uses examples from the `demo_module_7.c` code to illustrate each step. The code contains modifications to code templates that were produced by using `mib2c` on a MIB group in `SDK-DEMO1-MIB.txt`.

The `demo_module_7` example registers new instances as contexts that represent files. Subsequent `snmpget` requests to these contexts retrieve the size of a specified file.

▼ How to Enable Dynamic Update of a Multi-Instance Module

Steps

1. **Define two objects in the MIB for the module:**

   - A string with read-write `MAX-ACCESS` that, when set, registers the specified string as a context name.
   - A string with read-write `MAX-ACCESS` that, when set, unregisters the specified string context name.

   For example, the following objects, which are defined in the `SDK-DEMO1-MIB.txt` file, register and unregister a context string that is set with an `snmpset` request:

```
 me1createContext OBJECT-TYPE
    SYNTAX      OCTET STRING (SIZE(0..1024))
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
                "String which when set, registers a context."
    ::= { me1MultiGroup 2 }

  me1removeContext OBJECT-TYPE
    SYNTAX      OCTET STRING (SIZE(0..1024))
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
                "String which when set, unregisters a context."
    ::= { me1MultiGroup 3 }
```

2. **In the module, declare the location within the MIB tree where the OIDs for the context objects should be registered.**

   For example, the following code declares the OIDs for context strings:

```
// Registers a context
   static oid me1createContext_oid[] =
  { 1,3,6,1,4,1,42,2,2,4,4,6,1,2,0 };

   // Unregisters a context
   static oid me1removeContext_oid[] =
  { 1,3,6,1,4,1,42,2,2,4,4,6,1,3,0 };
```

3. **In the module, register both OIDs of the context objects with the SMA.**

   The following code shows an example:

```
    // Create a read-write registration handler named filesize,
    // which calls the set_createContext function to service snmp requests
    // for the me1createContext_oid object.  The OID_LENGTH argument
    // calculates the length of the me1createContext_oid.
    myreg1 = netsnmp_create_handler_registration
                                   ("filesize",
         set_createContext,
         me1createContext_oid,
        OID_LENGTH(me1createContext_oid),
        HANDLER_CAN_RWRITE);


    // Create a read-write registration handler named filesize,
    // which calls the set_removeContext function to service snmp requests
    // for the me1removeContext_oid object.  The OID_LENGTH argument
    // calculates the length of the me1removeContext_oid.
    myreg1 = netsnmp_create_handler_registration
        ("filesize",
         set_removeContext,
         me1removeContext_oid,
         OID_LENGTH(me1removeContext_oid),
         HANDLER_CAN_RWRITE);
```

4. **In the `set_createContext()` function handler code, extract the context name string from the SNMP message. Register the string as a new context.**

   The following code shows an example:

```
int
set_createContext(netsnmp_mib_handler *handler,
      netsnmp_handler_registration *reginfo,
      netsnmp_agent_request_info *reqinfo,
      netsnmp_request_info *requests)
{

// This handler handles set requests on the m1createContext_oid.
// The handler extracts the string from the snmp set request and
// uses it to register a new context for the me1filesize_oid.
//
// For detailed info. on net-snmp set processing,
// see http://www.net-snmp.org/tutorial-5/toolkit/mib_module/index.html
// The agent calls each SNMP mode in sequence. We include a case
// statement with only a break statement for each snmp set mode the
// the agent handles.  In this example, we implement only the
// snmp set action mode.  The case statement
// transfers control to the default: case when no other condition
// is satisfied.
netsnmp_handler_registration *myreg;
char *context_names[256];

switch(reqinfo->mode) {

case MODE_SET_RESERVE1:
            break;
case MODE_SET_RESERVE2:
            break;
case MODE_SET_FREE:
            break;
case MODE_SET_ACTION:

    // You must allocate memory for this variable because
    // the unregister_mib function frees it.
    filename = malloc(requests->requestvb->val_len + 1);
    snprintf(filename, sizeof(filename), "%s",  (u_char *)
                requests->requestvb->val.string);

    // Create a registration handler for the me1filesize_oid
    // object in the new context name specified by
    // the snmp set on the me1createContext OID.
    myreg = netsnmp_create_handler_registration
             ("test",
             get_test,
             me1filesize_oid,
             OID_LENGTH(me1filesize_oid),
             HANDLER_CAN_RONLY);
    myreg->contextName=filename;
    break;
case MODE_SET_COMMIT:
            break;
case MODE_SET_UNDO:
            break;

default:
    /* we should never get here, so this is a really bad error */
    DEBUGMSGTL(("filesize", "default CALLED\n"));
}
    return SNMP_ERR_NOERROR;
}
```

5. **In the `set_removeContext` handler code, extract the context name string from the SNMP message. Unregister the context.**

   The following code shows an example:

```
// This handler handles set requests on the m1removeContext_oid
int
set_removeContext(netsnmp_mib_handler *handler,
      netsnmp_handler_registration *reginfo,
      netsnmp_agent_request_info *reqinfo,
      netsnmp_request_info *requests)
{

    static int PRIORITY = 0;
    static int SUB_ID = 0;
    static int RANGE_UBOUND = 0;

    switch(reqinfo->mode) {

        case MODE_SET_RESERVE1:
            break;
        case MODE_SET_RESERVE2:
            break;
        case MODE_SET_ACTION:

       snprintf(filename, sizeof(filename), "%s\n",  (u_char *)
             requests->requestvb->val.string);
  unregister_mib_context(me1filesize_oid, OID_LENGTH(me1filesize_oid),
       PRIORITY, SUB_ID, RANGE_UBOUND,
       filename);
       break;
```

```
            case MODE_SET_COMMIT:
                break;
            case MODE_SET_FREE:
                break;
            case MODE_SET_UNDO:
                break;

     default:
         /* we should never get here, so this is a really bad error */
         DEBUGMSGTL(("filesize", "set_removeContext CALLED\n"));
     }
   return SNMP_ERR_NOERROR;
}
```

6. **In the handler code for a new context, get the context string from the `reginfo->contextName` variable.**

```
 /* This handler is called to handle snmp get requests for
       the me1filesize_oid for a specified context name. */

int
get_test(netsnmp_mib_handler *handler,
      netsnmp_handler_registration *reginfo,
      netsnmp_agent_request_info *reqinfo,
      netsnmp_request_info *requests)
{

/* We are never called for a GETNEXT if it's registered as an
   "instance", as it's "magically" handled for us.  */

/* An instance handler also only hands us one request at a time, so
   we don't need to loop over a list of requests; we'll only get one. */

struct stat buf;
static int fd = 0;

switch(reqinfo->mode) {

case MODE_GET:

   if (strcmp(reginfo->contextName, filename) == 0)

     // An open() for reading only returns without delay.
     if ((fd = open(filename, O_NONBLOCK | O_RDONLY)) == -1)
        DEBUGMSGTL(("filesize", "ERROR\n"));

     if (fstat(fd, &buf) == -1)
       DEBUGMSGTL(("filesize", "ERROR\n"));
     else
       DEBUGMSGTL(("filesize", "FILE SIZE IN BYTES = %d:\n", buf.st_size));

     snmp_set_var_typed_value(requests->requestvb, ASN_INTEGER,
     (u_char *) &buf.st_size /* XXX: a pointer to the scalar's data */,
     sizeof(buf.st_size) /* XXX: the length of the data in bytes */);
```

```
         break;

      default:
       /* we should never get here, so this is a really bad error */
          return SNMP_ERR_GENERR;
   }

   return SNMP_ERR_NOERROR;
   }
```

Registering New Instances in the Module

The demo_module_7 code example module registers context name strings that represent files. GET requests to these contexts retrieve the size of the file.

You do not need to edit the module to register new instances. The module can be dynamically updated to register new instances through the snmpset command. A management application passes the file name to the module by issuing an snmpset command, of the following format:

```
/usr/sfw/bin/snmpset -v 3 -u username -l authNoPriv -A "password" \
 hostname createContext_OID s "filename"
```

For example, the register_file script in the demo_module_7 directory issues a command that registers the file /usr/sfw sbin/snmpd as a new context name with the module:

```
/usr/sfw/bin/snmpset -v 3 -u myuser -l authNoPriv \
-A "mypassword" localhost .1.3.6.1.4.1.42.2.2.4.4.6.1.2.0 \
s "/usr/sfw/sbin/snmpd"
```

The module registers the set_createContext handler to handle incoming snmpset requests for the specified OID. The set_createContext handler registers the new file name as a context string in the contextName member of the netsnmp_registration_handler struct for the me1filesize_oid.

A management application can request the size of the file in blocks by issuing an snmpget command of the following format:

```
/usr/sfw/bin/snmpget -v 3 -u username -n contextname\
 -l authNoPriv -A "password" hostname me1filesize_oid
```

For example, the get_filesize script in the demo_module_7 directory issues a command that is similar to the following command:

```
/usr/sfw/bin/snmpget -m+SDK-DEMO6-MIB -v 3 -u myuser \
-n "/usr/sfw/sbin/snmpd" -l authNoPriv -A "mypassword" localhost \
.1.3.6.1.4.1.42.2.2.4.4.6.1.1.0
```

CHAPTER 8

Long–Running Data Collection

This chapter discusses the ways that you can enable a module to collect data over a long period of time without blocking the System Management Agent. The demonstration modules `demo_module_9` and `demo_module_10` illustrate these approaches.

This chapter contains the following topics:

About Long-Running Data Collection

SNMP is not ideally suited to collecting data that is generated over a period of time. Time-outs specified by an SNMP manager are generally only a few seconds, to enable most problems to be detected quickly. However, some data might be useful when looked at over a longer period, for example, to indicate a developing condition. Such data can only be collected through a *long-running data collection* to get around the timeout issue. You can code your module to perform long-running data collection. You can choose from several different design patterns to model such operations.

The following design patterns can be used to enable a module to handle long-running data collections through the agent.

SNMP alarm-based approach	The module registers an SNMP alarm to call a function at a specified interval. For most sites, this solution is most useful for performing long-running data collections. See "SNMP Alarm Method for Data Collection" on page 86 for more information and code examples.

SNMP manager polling	The SNMP manager polls a status variable to find out whether a data collection is complete, and to determine the age of the data. The data is retrieved when the status variable returns an acceptable value. The polling approach is most useful if your site has one SNMP manager and several SNMP agents. See "SNMP Manager Polling Method for Data Collection" on page 88 for more information and code examples.

SNMP Alarm Method for Data Collection

In the SNMP alarm method for long-running data collection, the module registers an SNMP alarm to call a function at a specified interval. The interval is specified in seconds. The function can be called one time, or called repeatedly until the alarm is unregistered. The module sets a flag that causes the agent to delegate the SNMP request. By delegating a request, the agent avoids blocking other requests while responding to a request. The agent caches the SNMP request information to be retrieved later when the request is handled. The `demo_module_9` example demonstrates the SNMP-alarm-based approach.

`demo_module_9` Code Example for SNMP Alarm Method

The `demo_module_9` code is located by default in `/usr/demo/sma_snmp/demo_module_9`. The `README_demo_module_9` file within that directory contains instructions that describe how to perform the following tasks:

- Compile source files to generate a shared library object that implements a module
- Set up the agent to dynamically load the module
- Test the module with `snmp` commands to show that the module is functioning as expected

The `demo_module_9` example implements the objects defined in the `SDK-DEMO9-MIB.txt`. The module demonstrates how to implement objects that normally would block the agent as the agent waits for external events. The agent can continue responding to other requests while this implementation waits.

This example uses the following features:

- Sets the `delegated` member of the `requests` structure to 1 to indicate to the agent that this request should be delayed. The agent queues this request to be handled later and then is available to handle other requests. The agent is not blocked by this request.
- Registers an SNMP alarm to update the results at a later time.

Managing the Timing of Data Collection

An important aspect of the `demo_module_9` example is the relationship between the SNMP timeout and the delay time interval of the module. The delay time interval is the interval in seconds after which the agent sends an alarm to the module. The `delay_time` variable in the module stores this value. By default, the delay time is set to 1 in the module. You can change this value by issuing an `snmpset` command on the `delayedInstanceOid` object and supplying an integer value. The `set_demo_module_9` script does issue the `snmpset` command to change the delay time interval. The new time interval value is used by the module to register for an alarm with the agent.

The agent calls the module when a `snmpget` or `snmpset` is issued on the `delayedInstanceOid` object. Instead of returning the requesting data right away, the module sets a flag to tell the agent that the request processing might take a while. The agent is free to handle other requests. The module then registers an alarm with the agent. The module needs some way to get the agent to return to the module and return the requested data when the data collection has completed. In `demo_module_9`, a one-time alarm is set to go off in 1 second. If you want a longer data collection, you can set the `delay_time` value to a longer interval. You can also set the alarm to go off repeatedly at a specified interval.

The module registers the alarm with a callback function. At the specific alarm interval, the agent calls the callback function in the module. In `demo_module_9`, the callback function is `return_delayed_response()`, which actually handles the SNMP GET or SNMP SET request.

The client that requested the data with SNMP GET must wait for the response from the agent. The `snmpget` command and other Net-SNMP tools have a default timeout value of 5 seconds. The client is likely to time out before getting the requested response. For this reason, you should increase the timeout value for the `snmpget` and `snmpset` commands.

You should increase the timeout of the command the amount of time required to complete the data collection. If you are doing an `snmpset`, make the timeout value 3 or 4 times longer than the delay time interval. A longer timeout is needed because a SET operation is more time-consuming than a GET. The agent makes several calls to the module to process a single SET, and each call is delayed by the delay value.

The `-t` option is used to set the timeout value. See the`snmpcmd`(1M) man page for more information about common command-line options for Net-SNMP tools.

SNMP Manager Polling Method for Data Collection

In the SNMP manager polling method, an SNMP manager polls a status variable to find out whether a data collection is complete. When the data collection is complete, the age of the data is determined. If the date of the data is not acceptable, the manager can set the status variable to start a new collection. The polling method is recommended if you have one SNMP manager that is to control the polling of one or more agents. The `demo_module_10` example demonstrates the SNMP manager polling approach.

`demo_module_10` Code Example for SNMP Polling Method

The `demo_module_10` code is located by default in `/usr/demo/sma_snmp/demo_module_10`. The `README_demo_module_10` file within that directory contains instructions that tell how to perform the following tasks:

- Compile source files to generate a shared library object that implements a module
- Set up the agent to dynamically load the module
- Test the module with `snmp` commands to show that the module is functioning as expected

The `demo_module_10` example implements the objects defined in the `SDK-DEMO10-MIB.txt`. The module is designed to handle long-running data collections so that their values can be polled by an SNMP manager. The module also shows how to implement objects that normally would block the agent as the agent waits for external events. The agent can continue responding to other requests while this implementation waits.

The `demo_module_10` module uses the following features:

- Sets the `delegated` member of the `requests` structure to 1 to indicate to the agent that this request should be delayed. The agent queues this request to be handled later and then is available to handle other requests. The agent is not blocked by this request.

- Registers an SNMP alarm to update the results at a later time.
- Uses `status` variable to communicate the status of a data collection to the polling SNMP manager.
- Uses `refreshTime` variable to return the date and time that the data collection completed.

Avoiding a Race Condition When Polling

A race condition can occur with two or more management applications. When multiple applications issue GET or SET protocol operations that span more than a single PDU, competition for the results occurs. In the case of a long-running data collection, a race condition can occur when the module completes data collection. The module updates the `status` variable to indicate that the data is ready to send. However, the agent issues a second GET operation on the same variable before the first request receives the requested data. If the module starts a new data collection in response to the second request, no data is available to return to the first request.

In the following figure, Mgr2's request is received by the module after Mgr1's request but before Mgr1 gets the data. This situation could happen if the module starts a new data collection while requests are pending.

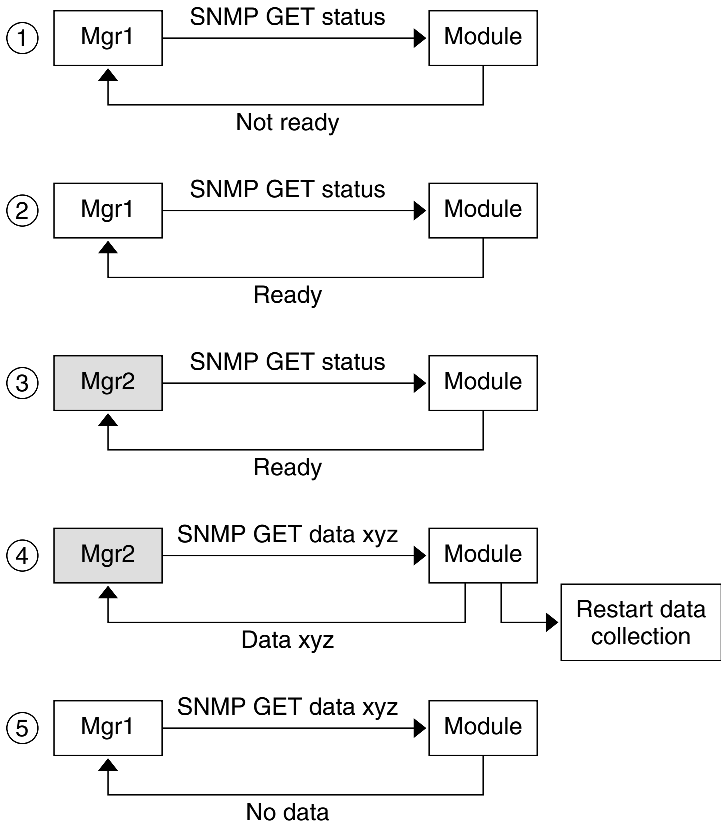

FIGURE 8–1 Race Condition When Polling for Data

To avoid this scenario, a module can define a flag to maintain the state of outstanding requests. When an SNMP request is received, the module checks the flag. The module starts a new collection only if no SNMP requests are outstanding. The module returns an SNMP error if requests are outstanding.

CHAPTER 9

Entity MIB

This chapter describes the implementation of the Entity MIB and the associated API functions in the System Management Agent. The demonstration module `demo_module_11` is used to explain how to use the MIB and the tables that are defined in the MIB. The chapter contains the following topics:

About the Entity MIB

The Entity MIB is defined by the Internet Engineering Task Force RFC 2737 at `http://www.ietf.org/rfc/rfc2737.txt`. This chapter does not describe the Entity MIB in detail. You should read RFC 2737 before reading this chapter.

The Entity MIB provides a mechanism for presenting hierarchies of physical entities by using SNMP tables. The Entity MIB contains the following groups, which describe the physical elements and logical elements of a managed system:

`entityPhysical` group

The `entityPhysical` group describes the identifiable physical resources that are managed by the agent. Resources include the chassis, boards, power supplies, sensors, and so on.

Physical entities are represented by rows in the `entPhysicalTable`, where one row is provided for each hardware resource. The rows are called *entries*. A particular row is referred to as an *instance*. Each table

	entry has a unique index, `entPhysicalIndex`, and contains several objects that represent common characteristics of the hardware resource. One object, `entPhysicalContainedIn`, points to the index of another row in this table. This object is used to indicate whether an entity is contained within another entity. A row for a system board might use `entPhysicalContainedIn` to specify the index of the row that represents the chassis where the board is installed.
`entityLogical` group	The `entityLogical` group describes the logical entities managed by the agent. Logical entities represent nonphysical, abstract elements that provide services. The abstract elements are controlled by higher levels of management. For example, logical entities might represent elements of platform hardware management. Such elements might include functions such as OS reboot, hardware reset, and power control. Logical entities might also represent administrative domains such as Solaris domains or service controllers. Logical entities are represented as rows in the `entLogicalTable`, which provides one row for each logical entity. Each table row has a unique index, `entLogicalIndex`, and contains objects for the logical entity's name, description, and type. Each row also contains security information that is applicable to SNMPv1, SNMPv2c, and SNMPv3 to allow access to the logical entity's MIB information. If an agent represents multiple logical entities with this MIB, the agent must implement the `entityLogical` group for all logical entities that are known to the agent. If an agent represents one logical entity, or multiple logical entities within a single naming scope, the agent can omit implementation of this group.
`entityMapping` group	The `entityMapping` group describes the objects that represent the associations between elements for which a single agent provides management information. These elements include multiple logical entities, physical components, interfaces, and port identifiers. The `entityMapping` group contains the following tables:

	■ The `entPhysicalContainsTable` provides a hierarchy of the hardware resources that are represented in the `entPhysicalTable`. The `entPhysicalContainsTable` table is two-dimensional, indexed first by the `entPhysicalIndex` of the containing entry, and second by the `entPhysicalChildIndex` of the contained entries. ■ The `entLPMappingTable` is the logical-physical mapping table. The `entLPMappingTable` makes associations between logical entities and physical entities by mapping the indexes of the `entLogicalTable` to the indexes of `entPhysicalTable` For example, the table could map a firewall to a particular board. ■ The `entAliasMappingTable` represents mappings of logical entity and physical component to external MIB identifiers.
`entityGeneral` group	This table describes objects that represent general entity information for which a single agent provides management information. Currently, only one object exists in this group. The object records the time interval between agent startup and the last change to the Physical Entity Table or Physical Mapping Table.

The RFC 2737 and the `ENTITY-MIB.txt` file describe these tables in more detail. The `ENTITY-MIB.txt` file is located in the `/etc/sma/snmp/mibs` directory.

SMA Entity MIB Implementation

The System Management Agent provides a module called `libentity.so` for use with the Entity MIB. This module is contained in the `/usr/sfw/include` directory.

The `libentity.so` module performs the following tasks when loaded:

- Registers OIDs for the Entity MIB
- Creates empty tables for the groups described and defined in RFC 2737
- Handles the rules and constraints of the Entity MIB tables and maintains table integrity as specified in RFC 2737
- Provides Entity API functions that support a module's ability to add, delete, and modify objects in the OID space of the Entity MIB

If you want your module to use the Entity MIB, you must load the `libentity.so` module into the agent before you load your module.

Using the Entity MIB

To use the Entity MIB, you must write a module to create objects that reflect the devices that you want to manage. You use the objects to populate the empty tables that are created by the `libentity.so` module. Your module must use the API functions that are documented in "Entity MIB API" on page 95. Use `demo_module_11`, which is described in "`demo_module_11` Code Example for Entity MIB" on page 117, to see how that module uses the API functions. The `demo_module_11` also contains table header files that you need to use the API functions. See "Header Files for Entity MIB Functions" on page 113.

After you write your module, you can use the following procedure to set up the agent to use the Entity MIB and your module.

▼ How to Set Up the Agent to Use the Entity MIB

Steps

1. **As root, add the appropriate `dlmod` statement for your operating system in the agent's configuration file `/etc/sma/snmp/snmpd.conf`.**

   - **On a 64-bit Solaris Operating System on SPARC:**

     ```
     dlmod entity /usr/sfw/lib/sparcv9/libentity.so
     ```

   - **On a 32-bit Solaris Operating System:**

     ```
     dlmod entity /usr/sfw/lib/libentity.so
     ```

2. **In the `/etc/sma/snmp/snmpd.conf` file, insert a `dlmod` statement for your module *after* the `dlmod` statement for the `libentity.so`.**

   For example, suppose your module is named `libacmerouter.so`. The module is located in `/home/username/lib`. You would enter the following line:

   ```
   dlmod acmerouter /home/username/lib/libacmerouter.so
   ```

   Your module must be loaded after the `entity` module because your module is dependent upon the `entity` module.

3. **Restart the SNMP agent.**

   ```
   # /etc/init.d/init.sma restart
   ```

Entity MIB API

This section lists and describes the API functions that are provided in the `libentity.so` module. Use these functions in your module when you want to use the Entity MIB.

TABLE 9–1 Entity MIB Functions Listed by Category

Function Category	Functions
"Physical Table Functions" on page 96	`allocPhysicalEntry()`
	`getPhysicalEntry()`
	`deletePhysicalTableEntry()`
	`makePhysicalTableEntryStale()`
	`makePhysicalTableEntryLive()`
	`getPhysicalStaleEntry()`
	`getAllChildrenFromPhysicalContainedIn()`
"Physical Contains Table Functions" on page 101	`addPhysicalContainsTableEntry()`
	`deletePhysicalContainsTableEntry()`
	`deletePhysicalContainsParentIndex()`
	`deletePhysicalContainsChildIndex()`
	`getPhysicalContainsChildren()`
"Logical Table Functions" on page 104	`allocLogicalEntry()`
	`getLogicalTableEntry()`
	`deleteLogicalTableEntry()`
	`makeLogicalTableEntryStale()`
	`makeLogicalTableEntryLive()`
	`getLogicalStaleEntry()`
"LP Mapping Table Functions" on page 108	`addLPMappingTableEntry()`
	`deleteLPMappingTableEntry()`
	`deleteLPMappingLogicalIndex()`
	`deleteLPMappingPhysicalIndex()`

TABLE 9–1 Entity MIB Functions Listed by Category *(Continued)*

Function Category	Functions
"Alias Mapping Table Functions" on page 111	`addAliasMappingTableEntry()`
	`deleteAliasMappingTableEntry()`
	`deleteAliasMappingLogicalIndex()`
	`deleteAliasMappingPhysicalIndex()`

Physical Table Functions

The `entPhysicalTable` contains one row for each physical entity. The table contains at least one row for an overall physical entity. Each table entry provides objects to help an NMS to identify and characterize the entry. Other objects in the table entry help an NMS to relate the particular entry to other entries in the table.

The following functions are for use with the `entPhysicalTable` in the Entity MIB.

- "`allocPhysicalEntry()`" on page 96
- "`getPhysicalEntry()`" on page 97
- "`deletePhysicalTableEntry()`" on page 98
- "`makePhysicalTableEntryStale()`" on page 98
- "`makePhysicalTableEntryLive()`" on page 99
- "`getPhysicalStaleEntry()`" on page 100
- "`getAllChildrenFromPhysicalContainedIn()`" on page 101

`allocPhysicalEntry()`

Synopsis

extern int **allocPhysicalEntry**(int *physidx*, entPhysicalEntry_t **newPhysEntry*);

Description

Allocates an entry in the `entPhysicalTable`. The *physidx* parameter is the requested physical index. If *physidx*= 0, the function tries to use the first available index in the table. If *physidx*= 1 or greater, the function tries to use the specified index. If the specified index is in use, the function returns the first available index in the table. As a result, the returned index might not be the same as the requested physical index.

The memory that is associated with *newPhysEntry* can be freed. The function creates an internal copy of the data.

The `entPhysicalEntry_t` structure definition is shown in "`entPhysicalEntry_t` Structure" on page 114. Special cases for *newPhysEntry* values are handled as shown in the following table.

Object	**Value of** *newPhysEntry*	**Entity MIB module handling**
`entPhysicalDescr`	NULL	reject
`entPhysicalVendorType`	NULL	`{ 0, 0 }`
`entPhysicalName`	NULL	`""`
`entPhysicalHardwareRev`	NULL	`""`
`entPhysicalFirmwareRev`	NULL	`""`
`entPhysicalSoftwareRev`	NULL	`""`
`entPhysicalSerialNum`	NULL	`""`
`entPhysicalMfgName`	NULL	`""`
`entPhysicalModelName`	NULL	`""`
`entPhysicalAlias`	NULL	`""`
`entPhysicalAssetID`	NULL	`""`

Returns

index	allocated to the physical entry.
`-1`	if an error occurs when adding the entry. Check the log for more details.

`getPhysicalEntry()`

Synopsis

```
entPhysicalEntry_t *getPhysicalEntry(int index);
```

Description

Gets the actual physical table entry for the specified index. The caller must *not* change the values or release the memory of the entry that is returned. The `entPhysicalEntry_t` structure definition is shown in "`entPhysicalEntry_t` Structure" on page 114.

Returns

`getPhysicalEntry()` returns the entry for the specified index.

Returns `NULL` if an error occurs while finding the entry, or if a stale entry exists. In this context, stale means that the entry details are present in the agent memory but should not be displayed during any SNMP operation.

`deletePhysicalTableEntry()`

Synopsis

`extern int` **`deletePhysicalTableEntry`**`(int` *xPhysicalIndex*`);`

Description

Deletes the physical table entry that is associated with the specified *xPhysicalIndex*. The instances of *xPhysicalIndex* in the `entAliasMappingTable`, `entLPMappingTable` and the `entPhysicalContainsTable` are also deleted to maintain integrity among the various Entity MIB tables.

Returns

`0`	for success.
`-1`	if the *xPhysicalIndex* is not found.
`-2`	if a stale entry was found for the *xPhysicalIndex*. In this context, "stale" means that the entry details are present in the agent memory but are not displayed during any SNMP operation.

`makePhysicalTableEntryStale()`

Synopsis

`extern int` **`makePhysicalTableEntryStale`**`(int` *xPhysicalIndex*`);`

Description

Makes the physical table entry that is associated with the *xPhysicalIndex* become stale. In this context, “stale” means that the entry details are present in the agent memory but are not displayed during any SNMP operation. The index that was allocated to a stale entry is not allocated to another entry.

When you make an entry become stale, the instances of *xPhysicalIndex* in the `entAliasMappingTable`, `entLPMappingTable` and `entPhysicalContainsTable` are also deleted. The deletion maintains integrity among the various Entity MIB tables. Before you make an entry stale, you might want to store the entries that are to be deleted from the tables.

The physical table entry can be made available or “live” again by calling the `makePhysicalTableEntryLive()` functions, which is described in “`makePhysicalTableEntryLive()`” on page 99.

Returns

`0`	for success.
`-1`	if the *xPhysicalIndex* is not found.
`-2`	if a stale entry already exists for *xPhysicalIndex*.

`makePhysicalTableEntryLive()`

Synopsis

`extern int` **`makePhysicalTableEntryLive`**`(int` *xPhysicalIndex*`);`

Description

Makes the stale physical table entry associated with the *xPhysicalIndex* live. In this context, “live” means that the entry details that are present in the agent memory are displayed during SNMP operations. The entry can be made stale by calling the `makePhysicalTableEntryStale()` function. In this context, “stale” means that the entry details are present in the agent memory but are not displayed during any SNMP operation.

If a stale entry is made live again, you must recreate the corresponding entries that were deleted in the `entPhysicalContainsTable`, the `entLPMappingTable`, and the `entAliasMappingTable`. Use the appropriate functions for adding an entry to each table: "`addPhysicalContainsTableEntry()`" on page 101, "`addLPMappingTableEntry()`" on page 109, and "`addAliasMappingTableEntry()`" on page 111.

Returns

`0`	for success.
`-1`	if the *xPhysicalIndex* is not found.
`-2`	if a live entry already exists for *xPhysicalIndex*.

`getPhysicalStaleEntry()`

Synopsis

`entPhysicalEntry_t` **`*getPhysicalStaleEntry`**`(int` *index*`);`

Description

The caller must *not* change the values or release the memory of the entry that is returned.

Gets the stale physical table index structure for the specified index. In this context, stale means that the entry details are present in the agent memory but are not displayed during any SNMP operation. The `entPhysicalEntry_t` structure definition is shown in "`entPhysicalEntry_t` Structure" on page 114.

Returns

Returns the index structure for the specified index.

Returns `NULL` if an error occurs while finding the entry, or if a live entry exists.

`getAllChildrenFromPhysicalContainedIn()`

Synopsis

`int` **`getAllChildrenFromPhysicalContainedIn`**`(int` *parentIndex*`);`

Description

Gets the indexes for all children in the `entPhysicalTable` that have *parentIndex* as their parent in the `entPhysicalContainedIn` field.

Returns

Returns an array of integer indexes with null termination.

Returns `NULL` if no children, or invalid index, or not enough memory when allocating the array.

Physical Contains Table Functions

The `entPhysicalContainsTable` exposes the container relationships between physical entities. This table provides the same information that can be found by constructing the virtual containment tree for a given `entPhysicalTable`, but in a more direct format.

The following functions are for use with the `entPhysicalContainsTable` in the Entity MIB:

- "`addPhysicalContainsTableEntry()`" on page 101
- "`deletePhysicalContainsTableEntry()`" on page 102
- "`deletePhysicalContainsParentIndex()`" on page 103
- "`deletePhysicalContainsChildIndex()`" on page 103
- "`getPhysicalContainsChildren()`" on page 104

`addPhysicalContainsTableEntry()`

Synopsis

`extern int` **`addPhysicalContainsTableEntry`**`(int` *entPhysicalIndex*`, int` *childIndex*`);`

Description

Adds an entry to the `entPhysicalContainsTable` table for the specified *entPhysicalIndex* and *childIndex*. The `entPhysicalContainedIn` OID that is present in the `entPhysicalTable` for the *childIndex* might be replaced by the OID for *entPhysicalIndex*. The OID is replaced if the *entPhysicalIndex* has a lower index than the original index.

Returns

`0`	for successful addition.
`-1`	for failure to add.
`-2`	for stale index.
`1`	if the entry already exists for the specified *entPhysicalIndex* and *childIndex*.

`deletePhysicalContainsTableEntry()`

Synopsis

`extern int` **`deletePhysicalContainsTableEntry`**`(int` *parentIndex*`, int` *childIndex*`);`

Description

Deletes the *parentIndex* or *childIndex* entry that is present in the `entPhysicalContainsTable`.

Returns

`0`	for success.
`-1`	for failure.
`-2`	for stale entry, either parent or child, or both.

`deletePhysicalContainsParentIndex()`

Synopsis

`extern int` **`deletePhysicalContainsParentIndex`**`(int` *parentIndex*`);`

Description

Deletes all entries in the `entPhysicalContainsTable` where the parent index is equal to the specified *parentIndex*.

Returns

number	of children successfully deleted for the specified parent.
`-1`	for failure.
`-2`	for stale parent entry.

`deletePhysicalContainsChildIndex()`

Synopsis

`extern int` **`deletePhysicalContainsChildIndex`**`(int` *childIndex*`);`

Description

Deletes all entries in the `entPhysicalContains` table where the child index is equal to the specified *childIndex*.

Returns

number	of parents successfully deleted for the specified child.
`-1`	for failure.
`-2`	for stale child entry.

`getPhysicalContainsChildren()`

Synopsis

`extern int` **`getPhysicalContainsChildren`**`(int` *parentIndex*`);`

Description

Get the indexes for all the children of the specified parent in the `entPhysicalContainsTable`.

Returns

Returns an array of integer indexes, with null termination.

Returns `NULL` if children exist, or if not enough memory exists when allocating the array. The array is a copy that should be freed when done.

Logical Table Functions

The `entLogicalTable` table contains one row per logical entity. For agents that implement more than one naming scope, at least one entry must exist. Agents that instantiate all MIB objects within a single naming scope are not required to implement this table.

The following functions are for use with the `entLogicalTable` in the Entity MIB:

- "`allocLogicalEntry()`" on page 104
- "`deleteLogicalTableEntry()`" on page 106
- "`makeLogicalTableEntryStale()`" on page 106
- "`makeLogicalTableEntryLive()`" on page 107
- "`getLogicalStaleEntry()`" on page 108

`allocLogicalEntry()`

Synopsis

`extern int` **`allocLogicalEntry`**`(int` *logidx*`, entLogicalEntry_t`
`*`*xnewLogicalEntry*`);`

Description

Allocates an entry in the Logical Table. The *logidx* parameter is the requested logical index. If *logidx*= 0, the function tries to use the first available index in the table. If *logidx*= 1 or greater, the function tries to use the specified index. If the specified index is in use, the function returns the first available index in the table. As a result, the returned index might not be the same as the requested logical index.

The `allocLogicalEntry()` function returns the logical index that is allocated to the entry. The memory that is associated with *xnewLogicalEntry* can be freed. The function creates a internal copy of the data.

The `entLogicalEntry_t` structure definition is shown in "`entLogicalEntry_t` Structure" on page 114. Special cases for *xnewLogicalEntry* values are handled as shown in the following table.

Object	**Value of** *xnewLogicalEntry*	**Entity MIB module handling**
`entLogicalDescr`	NULL	reject
`entLogicalType`	NULL	`{ 1,3,6,1,2,1 }`
`entLogicalCommunity`	NULL	`""`
`entLogicalTAddress`	NULL or `""`	reject
`entLogicalTDomain`	NULL	reject
`entLogicalContextEngineId`	NULL	`""`
`entLogicalContextName`	NULL	`""`

Returns

Returns the index allocated to the logical entry.

Returns `-1` for error in adding the entry. Check the log for more details.

`getLogicalTableEntry()`

Synopsis

`entLogicalEntry_t` **`*getLogicalTableEntry`**`(int` *xLogicalIndex*`);`

Description

This function gets the logical table index structure for a particular index. The caller must not change the value or release the memory of the entry that is returned. The `entLogicalEntry_t` structure definition is shown in "`entLogicalEntry_t` Structure" on page 114.

Returns

Returns the entry that is associated with *xLogicalIndex*.

Returns `NULL` on error in finding the entry, or if a stale entry exists.

`deleteLogicalTableEntry()`

Synopsis

`extern int` **`deleteLogicalTableEntry`**`(int` *xLogicalIndex*`);`

Description

Deletes the logical table entry that is associated with the *xLogicalIndex*. The instances of *xLogicalIndex* in the `entAliasMappingTable` and the `entLPMappingTable` are also deleted to maintain integrity among the various Entity MIB tables.

Returns

`0`	for success.
`-1`	if the *xLogicalIndex* is not found.
`-2`	if a stale entry was found for *xLogicalIndex*.

`makeLogicalTableEntryStale()`

Synopsis

`extern int` **`makeLogicalTableEntryStale`**`(int` *xLogicalIndex*`);`

Description

Makes the logical table entry associated with the *xLogicalIndex* become stale. In this context, "stale" means that the entry details are present in the agent memory but are not displayed during any SNMP operation. The index that was allocated to a stale entry is not allocated to another entry.

When you make an entry become stale, the instances of *xLogicalIndex* in the `entAliasMappingTable`, `entLPMappingTable` and `entPhysicalContainsTable` are also deleted. The deletion maintains integrity among the various Entity MIB tables. Before you make an entry stale, you might want to store the entries that are to be deleted from the other tables.

The stale logical table entry can be made available again by calling the `makeLogicalTableEntryLive()` function, which is described in "`makeLogicalTableEntryLive()`" on page 107.

Returns

`0`	for success.
`-1`	if the *xLogicalIndex* is not found.
`-2`	if a stale entry was found for *xLogicalIndex*.

`makeLogicalTableEntryLive()`

Synopsis

extern int **makeLogicalTableEntryLive**(int *xLogicalIndex*);

Description

Makes the stale logical table entry associated with the *xLogicalIndex* become live. In this context, "live" means that the entry details that are present in the agent memory are displayed during any SNMP operations. The entry can be made stale by calling the `makeLogicalTableEntryStale()` function.

If an entry is made live again, you must recreate the corresponding entries that were deleted in the `entPhysicalContainsTable`, the `entLPMappingTable`, and the `entAliasMappingTable`. Use the appropriate functions for adding an entry to each table: "`addPhysicalContainsTableEntry()`" on page 101, "`addLPMappingTableEntry()`" on page 109, and "`addAliasMappingTableEntry()`" on page 111.

Returns

`0`	for success.
`-1`	if the *xLogicalIndex* is not found.
`-2`	if a live entry already exists for *xLogicalIndex*.

`getLogicalStaleEntry()`

Synopsis

`entLogicalEntry_t` **`*getLogicalStaleEntry`**`(int` *index* `);`

Description

Gets the stale logical table index structure for the specified index. The caller must not change the values or release the memory of the entry that is returned. The `entLogicalEntry_t` structure definition is shown in "`entLogicalEntry_t` Structure" on page 114.

Returns

Returns the stale entry for the specified index.

Returns `NULL` if the entry is not found, or if a live entry exists.

LP Mapping Table Functions

The `entLPMappingTable` contains zero or more rows that associate logical entities to physical equipment. For each logical entity that is known by this agent, there are zero or more mappings to the physical resources that are used to realize that logical entity. An agent should limit the number and nature of entries in this table so that only meaningful and non-redundant information is returned. See the `/etc/sma/snmp/mibs/ENTITY-MIB.txt` file for more information about the `entLPMappingTable`.

The following functions are for use with the `entLPMappingTable`:

- "`addLPMappingTableEntry()`" on page 109
- "`deleteLPMappingTableEntry()`" on page 109
- "`deleteLPMappingLogicalIndex()`" on page 110
- "`deleteLPMappingPhysicalIndex()`" on page 110

`addLPMappingTableEntry()`

Synopsis

`extern int` **`addLPMappingTableEntry`**`(int` *xentLogicalIndex*`, int` *xentPhysicalIndex* `);`

Description

Adds an entry to the `entLPMappingTable` with the *xentLogicalIndex* as the primary index and *xentPhysicalIndex* as the secondary index.

Returns

`0`	for successful addition.
`1`	if the entry already exists for the given *xentPhysicalIndex* and *xentLogicalIndex*.
`-1`	for failure to add.
`-2`	for stale index.

`deleteLPMappingTableEntry()`

Synopsis

`extern int` **`deleteLPMappingTableEntry`**`(int` *xentLogicalIndex*`, int` *xentPhysicalIndex* `);`

Description

Deletes the entry of the LP Mapping table that uses the specified *xentLogicalIndex* as the primary index and *xentPhysicalIndex* as the secondary index.

Returns

`0`	for successful deletion.
`-1`	for failure to delete.
`-2`	for stale entry, either logical index or physical index, or both.

`deleteLPMappingLogicalIndex()`

Synopsis

`extern int` **`deleteLPMappingLogicalIndex`**`(int` *xentLogicalIndex*`);`

Description

Deletes all the entries of the `entLPMappingTable` that have the *xentLogicalIndex* as the primary index.

Returns

number	of successfully deleted entries.
`-1`	for failure.
`-2`	for stale logical entry.

`deleteLPMappingPhysicalIndex()`

Synopsis

`extern int` **`deleteLPMappingPhysicalIndex`**`(int` *xentPhysicalIndex*`);`

Description

Deletes all the entries of the `entLPMappingTable` that have *xentPhysicalIndex* as the secondary index.

Returns

*number*of successfully deleted entries.

`-1`	if no entry was deleted.
`-2`	for stale physical entry.

Alias Mapping Table Functions

The `entAliasMappingTable` contains zero or more rows that represent mappings of logical entity and physical entities for ports to external MIB identifiers. Each physical port in the system can be associated with a mapping to an external identifier. The external identifier is associated with a particular logical entity's naming scope. A wildcard mechanism is provided to indicate that an identifier is associated with more than one logical entity.

The following functions are for use with the `entAliasMappingTable` in the Entity MIB:

- "`addAliasMappingTableEntry()`" on page 111
- "`deleteAliasMappingTableEntry()`" on page 112
- "`deleteAliasMappingLogicalIndex()`" on page 112
- "`deleteAliasMappingPhysicalIndex()`" on page 113

`addAliasMappingTableEntry()`

Synopsis

`extern int` **`addAliasMappingTableEntry`**`(int` *xentPhysicalIndex*`, int` *xentLogicalIndex*`, oid*` *xAliasMapId*`, int` *xAliasMapIdSize*`);`

Description

Adds an entry to the `entAliasMappingTable` with the *xentPhysicalIndex* as the primary index and *xentLogicalIndex* as the secondary index. *xAliasMapId* is the alias (OID) for the entry and *xAliasMapIdSize* is the size in bytes of *xAliasMapId*.

Note that if *entAliasMapId* = `NULL`, the request is rejected.

Returns

`0`	for successful addition.
`1`	if the entry already exists for the given *xentPhysicalIndex* and *xentLogicalIndex*.
`-1`	for failure.
`-2`	for stale entry.

`deleteAliasMappingTableEntry()`

Synopsis

`extern int` **`deleteAliasMappingTableEntry`**`(int` *xentPhysicalIndex*`, int` *xentLogicalIndex*`);`

Description

Deletes the entry in the `entAliasMappingTable` that has *xentPhysicalIndex* as the primary index and *xentLogicalIndex* as the secondary index.

Returns

`0`	for successful deletion.
`-1`	for entry not found.
`-2`	for stale entry.

`deleteAliasMappingLogicalIndex()`

Synopsis

`extern int` **`deleteAliasMappingLogicalIndex`**`(int` *xentLogicalIndex*`);`

Description

Deletes all entries of the `entAliasMappingTable` that have *xentLogicalIndex* as the secondary index.

This function cannot be used to delete all indexes that have an *xentLogicalIndex* of zero. Use the `deleteAliasMappingTableEntry()` function to delete such entries one at a time, with the appropriate *xentPhysicalIndex* specified.

Returns

number	of entries successfully deleted.
`-1`	for entry not found.

`-2`	for stale logical entry.

`deleteAliasMappingPhysicalIndex()`

Synopsis

`extern int` **`deleteAliasMappingPhysicalIndex`**`(int` *xentPhysicalIndex*`);`

Description

Deletes all entries in the `entAliasMappingTable` whose primary index matches the specified *xentPhysicalIindex*.

Returns

*number*of entries successfully deleted.

`-1`	for entry not found.
`-2`	for stale physical entry.

Header Files for Entity MIB Functions

Data declarations and defines that are needed by the Entity MIB functions are included in header files. The following header files in `/usr/demo/sma_snmp/demo_module_11` can be copied and modified for use with your own modules:

```
entAliasMappingTable.h
entLastChangeTime.h
entLogicalTable.h
entLPMappingTable.h
entPhysicalContainsTable.h
entPhysicalTable.h
```

The structures defined in `entPhysicalTable.h` and `entLogicalTable.h` are shown in the following sections.

entPhysicalEntry_t Structure

The entPhysicalTable.h header file contains the typedef for the entPhysicalEntry_t structure. This structure is representative of the entPhysicalTable columns that are defined in RFC 2737. The entPhysicalEntry_t is defined as follows:

```
typedef struct entPhysicalEntry_s {
    int_l entPhysicalIndex;
    char *entPhysicalDescr;
    oid *entPhysicalVendorType;
    int_l entPhysicalVendorTypeSize;
    int_l entPhysicalContainedIn;
    int_l entPhysicalClass;
    int_l entPhysicalParentRelPos;
    char *entPhysicalName;
    char *entPhysicalHardwareRev;
    char *entPhysicalFirmwareRev;
    char *entPhysicalSoftwareRev;
    char *entPhysicalSerialNum;
    char *entPhysicalMfgName;
    char *entPhysicalModelName;
    char *entPhysicalAlias;
    char *entPhysicalAssetID;
    int_l entPhysicalIsFRU;
    struct entPhysicalEntry_s *pNextEntry;
} entPhysicalEntry_t;
```

entLogicalEntry_t Structure

The entLogicalTable.h header file contains the typedef for the entLogicalEntry_tstructure. This structure is representative of the entLogicalTable columns that are defined in RFC 2737. The entLogicalEntry_t is defined as follows:

```
typedef struct entLogicalEntry_s {

   int_l   entLogicalIndex;
   char *entLogicalDescr;
   oid  *entLogicalType;
   int_l   entLogicalTypeSize;
   char *entLogicalCommunity;
   char *entLogicalTAddress;
   oid  *entLogicalTDomain;
   int_l   entLogicalTDomainSize;
   char *entLogicalContextEngineId;
   char *entLogicalContextName;
   struct entLogicalEntry_s* pNextEntry;

} entLogicalEntry_t;
```

Tips for Using Entity MIB Functions

Creating physical or logical entries	Create the appropriate physical entries or logical entries first, before creating the entries in the three mapping tables: `entLPMappingTable`, `entAliasMappingTable`, and the `entPhysicalContainsTable`.
Multiple parents	For physical entries that have more than one parent, all relationships must be defined in the `entPhysicalContainsTable`. For example, suppose you want to define that C is contained in A with the *entPhysicalContainedIn* field. You also want to define that C is also contained in B. In this case, you must define that C is contained in A, and C is contained in B in the `entPhysicalContainsTable`.
Recursive Relationships	Recursive relationships are not allowed in the `entPhysicalTable` and `entPhysicalContainsTable`. For example, suppose B is contained in A, and C is contained in B. In this case, A cannot be contained in C. The parent/child relationship is defined both in the `entPhysicalContainedIn` field of the `entPhysicalTable()` function and in the `entPhysicalContainsTable`. The recursive check safeguard is already built into the `addPhysicalContainsTableEntry()` function.
Uniqueness	When you specify *entPhysicalParentRelPos*, the `allocPhysicalEntry()` function does not check for uniqueness. For example, you can specify that A and B are contained in C by setting both *entPhysicalParentRelPos* fields to the same value. However, doing so would violate RFC 2737. The uniqueness of many fields is not necessarily checked by the functions. You must be aware of this fact during the design phase.
Deleting physical or logical entries	*Deleting* an entry is similar to making the entry *stale*. Both deleted and stale entries no longer show up in tables when performing SNMP operations. Whether you delete an entry or

	make an entry stale, the corresponding entries are automatically deleted in the three mapping tables. Note that you cannot undelete these corresponding mapping tables entries. This deletion is done to maintain the integrity of the tables. The difference between deleting an entry and making the entry stale is that a stale entry can be restored. Stale entries can be made *live* with functions that are designed for that purpose. A deleted entry cannot be restored.
Deleting Parents	The integrity of the `entPhysicalTable` and `entPhysicalContainsTable` are not maintained if you delete a parent before you delete the subsequent generations. The `deletePhysicalTableEntry()` function does not recursively remove the parent and its subsequent generations. The function only removes the specified entry from the tables. If you do not delete a parent's generations before deleting the parent, you leave orphaned children. This practice is a violation of RFC 2737. When you delete a parent of a multi-parent child, the *entPhysicalContainedIn* parameter is reset automatically to the lowest of the remaining parent index. RFC 2737 requires this reset. The *entPhysicalParentRelPos* parameter is then out of place. No API function lets you change that parameter. You can modify the *entPhysicalParentRelPos* parameter by manipulating the entry that is returned by the `getPhysicalTableEntry()` function. However, this approach for modifying *entPhysicalParentRelPos* is not supported. If you decide to try this approach, use caution.
Traps	A notification trap is sent out whenever a change is made to any of the five tables, such as the creation or deletion of entries. A mechanism exists to suppress traps from being sent too frequently. The throttling period is five seconds.

RFC Constraints and errors	The Entity MIB implementation has some constraints, which are dictated by RFC 2737. The only mechanism to notify the user about an error is through the error codes. You must understand the RFC thoroughly to be aware of the constraints.

`demo_module_11` Code Example for Entity MIB

The `/usr/demo/sma_snmp/demo_module_11` code example shows how the Entity MIB module can be used. The demo module is designed to populate the empty MIB tables that are created when the `libentity.so` module is dynamically loaded into the agent. The data that is loaded is described in this section.

You should examine the code in `demo_module_11`, especially the code in the `MyTable.c` file. The file `README_demo_module_11` in that directory includes procedures for building and using the example.

The `demo_module_11` example refers to a system with the following components that need to be managed:

- Two boards, with two CPU modules on each board
- One board that contains three ports
- Two logical domains
- Two firewall instances

These components can be divided into the following entities:

- 14 physical entities

  1 chassis
  3 slots in the chassis
  3 boards in the slots
  4 CPU modules in two boards
  3 ports in one board

- 4 logical entities

  2 domains
  2 firewalls

Some of the physical entities are contained in other physical entities. The logical entities are associated with particular physical entities. The Entity MIB tables should be populated to show the relationships among the various entities.

The following examples demonstrate how the MIB tables could be populated for this system.

EXAMPLE 9–1 Physical Entities for `demo_module_11`

The `entPhysicalTable` might be populated with the following values:

- One field-replaceable physical chassis:

```
entPhysicalDescr.1 ==              'Sun Chassis Model b1000'
entPhysicalVendorType.1 ==         sun.chassisTypes.1
entPhysicalContainedIn.1 ==        0
entPhysicalClass.1 ==              chassis(3)
entPhysicalParentRelPos.1 ==       -1
entPhysicalName.1 ==               'b1000'
entPhysicalHardwareRev.1 ==        'A(1.00.02)'
entPhysicalSoftwareRev.1 ==        ''
entPhysicalFirmwareRev.1 ==        ''
entPhysicalSerialNum.1 ==          'C100076544'
entPhysicalMfgName.1 ==            'Sun Microsystems'
entPhysicalModelName.1 ==          'CHS-1000'
entPhysicalAlias.1 ==              'cl-SJ17-3-006:rack1:rtr-U3'
entPhysicalAssetID.1 ==            '0007372293'
entPhysicalIsFRU.1 ==              true(1)
```

- Slot 1 within the chassis:

```
entPhysicalDescr.2 ==              'Sun Chassis Slot Type AA'
entPhysicalVendorType.2  ==        sun.slotTypes.1
entPhysicalContainedIn.2 ==        1
entPhysicalClass.2 ==              container(5)
entPhysicalParentRelPos.2 ==       1
entPhysicalName.2 ==               'S1'
entPhysicalHardwareRev.2 ==        'B(1.00.01)'
entPhysicalSoftwareRev.2 ==        ''
entPhysicalFirmwareRev.2 ==        ''
entPhysicalSerialNum.2 ==          ''
entPhysicalMfgName.2 ==            'Sun Microsystems'
entPhysicalModelName.2 ==          'SLT-AA97'
entPhysicalAlias.2 ==              ''
entPhysicalAssetID.2 ==            ''
entPhysicalIsFRU.2 ==              false(2)
```

- Slot 2 within the chassis:

```
entPhysicalDescr.3 ==              'Sun Chassis Slot Type AA'
entPhysicalVendorType.3 =          sun.slotTypes.1
entPhysicalContainedIn.3 ==        1
entPhysicalClass.3 ==              container(5)
entPhysicalParentRelPos.3 ==       2
```

EXAMPLE 9–1 Physical Entities for `demo_module_11` *(Continued)*

```
entPhysicalName.3 ==              'S2'
entPhysicalHardwareRev.3 ==       '1.00.07'
entPhysicalSoftwareRev.3 ==       ''
entPhysicalFirmwareRev.3 ==       ''
entPhysicalSerialNum.3 ==         ''
entPhysicalMfgName.3 ==           'Sun Microsystems'
entPhysicalModelName.3 ==         'SLT-AA97'
entPhysicalAlias.3 ==             ''
entPhysicalAssetID.3 ==           ''
entPhysicalIsFRU.3 ==             false(2)
```

- Slot 3 within the chassis:

```
entPhysicalDescr.4 ==             'Sun Chassis Slot Type AA'
entPhysicalVendorType.4 =         sun.slotTypes.1
entPhysicalContainedIn.4 ==       1
entPhysicalClass.4 ==             container(5)
entPhysicalParentRelPos.4 ==      3
entPhysicalName.4 ==              'S3'
entPhysicalHardwareRev.4 ==       '1.00.07'
entPhysicalSoftwareRev.4 ==       ''
entPhysicalFirmwareRev.4 ==       ''
entPhysicalSerialNum.4 ==         ''
entPhysicalMfgName.4 ==           'Sun Microsystems'
entPhysicalModelName.4 ==         'SLT-AA97'
entPhysicalAlias.4 ==             ''
entPhysicalAssetID.4 ==           ''
entPhysicalIsFRU.4 ==             false(2)
```

- Board 1 within Slot 1:

```
entPhysicalDescr.5 ==             'Sun CPU-100'
entPhysicalVendorType.5  ==       sun.moduleTypes.14
entPhysicalContainedIn.5 ==       2
entPhysicalClass.5 ==             module(9)
entPhysicalParentRelPos.5 ==      1
entPhysicalName.5 ==              'M1'
entPhysicalHardwareRev.5 ==       '1.00.07'
entPhysicalSoftwareRev.5 ==       '1.5.1'
entPhysicalFirmwareRev.5 ==       'A(1.1)'
entPhysicalSerialNum.5 ==         'C100087363'
entPhysicalMfgName.5 ==           'Sun Microsystems'
entPhysicalModelName.5 ==         'R10-FE00'
entPhysicalAlias.5 ==             'rtr-U3:m1:SJ17-3-eng'
entPhysicalAssetID.5 ==           '0007372562'
entPhysicalIsFRU.5 ==             true(1)
```

EXAMPLE 9–1 Physical Entities for `demo_module_11` *(Continued)*

- First CPU, in Board 1, within Slot 1:

```
entPhysicalDescr.6 ==              'Sun Ultrasparc-III 400MHz'
entPhysicalVendorType.6  ==        sun.cpuTypes.2
entPhysicalContainedIn.6 ==        5
entPhysicalClass.6 ==              other(1)
entPhysicalParentRelPos.6 ==       1
entPhysicalName.6 ==               'P1'
entPhysicalHardwareRev.6 ==        'G(1.02)'
entPhysicalSoftwareRev.6 ==        ''
entPhysicalFirmwareRev.6 ==        '1.1'
entPhysicalSerialNum.6 ==          ''
entPhysicalMfgName.6 ==            'Sun Microsystems'
entPhysicalModelName.6 ==          'SFE-400M'
entPhysicalAlias.6 ==              ''
entPhysicalAssetID.6 ==            ''
entPhysicalIsFRU.6 ==              false(2)
```

- Second CPU, in Board 1, within Slot 1:

```
entPhysicalDescr.7 ==              'Sun Ultrasparc-III 400MHz'
entPhysicalVendorType.7  ==        sun.cpuTypes.2
entPhysicalContainedIn.7 ==        5
entPhysicalClass.7 ==              other(1)
entPhysicalParentRelPos.7 ==       2
entPhysicalName.7 ==               'P2'
entPhysicalHardwareRev.7 ==        'G(1.02)'
entPhysicalSoftwareRev.7 ==        ''
entPhysicalFirmwareRev.7 ==        '1.1'
entPhysicalSerialNum.7 ==          ''
entPhysicalMfgName.7 ==            'Sun Microsystems'
entPhysicalModelName.7 ==          'SFE-400M'
entPhysicalAlias.7 ==              ''
entPhysicalAssetID.7 ==            ''
entPhysicalIsFRU.7 ==              false(2)
```

- Board 2 within Slot 2:

```
entPhysicalDescr.8 ==              'Sun CPU-200'
entPhysicalVendorType.8  ==        sun.moduleTypes.15
entPhysicalContainedIn.8 ==        3
entPhysicalClass.8 ==              module(9)
entPhysicalParentRelPos.8 ==       1
entPhysicalName.8 ==               'M2'
entPhysicalHardwareRev.8 ==        '2.01.00'
entPhysicalSoftwareRev.8 ==        '3.0.7'
entPhysicalFirmwareRev.8 ==        'A(1.2)'
```

```
entPhysicalSerialNum.8 ==         'C100098732'
entPhysicalMfgName.8 ==           'Sun Microsystems'
entPhysicalModelName.8 ==         'R10-FE0C'
entPhysicalAlias.8 ==             'rtr-U3:m2:SJ17-2-eng'
entPhysicalAssetID.8 ==           '0007373982'
entPhysicalIsFRU.8 ==             true(1)
```

- Third CPU, in Board 2, within Slot 2:

```
entPhysicalDescr.9 ==             'Sun Ultrasparc-III 400MHz'
entPhysicalVendorType.9 ==        sun.cpuTypes.5
entPhysicalContainedIn.9 ==       8
entPhysicalClass.9 ==             other(1)
entPhysicalParentRelPos.9 ==      1
entPhysicalName.9 ==              'P3'
entPhysicalHardwareRev.9 ==       'CC(1.07)'
entPhysicalSoftwareRev.9 ==       '2.0.34'
entPhysicalFirmwareRev.9 ==       '1.1'
entPhysicalSerialNum.9 ==         ''
entPhysicalMfgName.9 ==           'Sun Microsystems'
entPhysicalModelName.9 ==         'SFE-400M'
entPhysicalAlias.9 ==             ''
entPhysicalAssetID.9 ==           ''
entPhysicalIsFRU.9 ==             false(2)
```

- Fourth CPU, in Board 2, within Slot 2:

```
entPhysicalDescr.10 ==            'Sun Ultrasparc-III 400MHz'
entPhysicalVendorType.10 ==       sun.cpuTypes.2
entPhysicalContainedIn.10 ==      8
entPhysicalClass.10 ==            other(1)
entPhysicalParentRelPos.10 ==     2
entPhysicalName.10 ==             'P4'
entPhysicalHardwareRev.10 ==      'G(1.04)'
entPhysicalSoftwareRev.10 ==      ''
entPhysicalFirmwareRev.10 ==      '1.3'
entPhysicalSerialNum.10 ==        ''
entPhysicalMfgName.10 ==          'Sun Microsystems'
entPhysicalModelName.10 ==        'SFE-400M'
entPhysicalAlias.10 ==            ''
entPhysicalAssetID.10 ==          ''
entPhysicalIsFRU.10 ==            false(2)
```

- Board 3 within Slot 3:

EXAMPLE 9–1 Physical Entities for `demo_module_11` *(Continued)*

```
entPhysicalDescr.11 ==             'Sun port-200'
entPhysicalVendorType.11  ==       sun.moduleTypes.25
entPhysicalContainedIn.11 ==       4
entPhysicalClass.11 ==             module(9)
entPhysicalParentRelPos.11 ==      1
entPhysicalName.11 ==              'M2'
entPhysicalHardwareRev.11 ==       '2.01.00'
entPhysicalSoftwareRev.11 ==       '3.0.7'
entPhysicalFirmwareRev.11 ==       'A(1.2)'
entPhysicalSerialNum.11 ==         'C100098732'
entPhysicalMfgName.11 ==           'Sun Microsystems'
entPhysicalModelName.11 ==         'R11-C100'
entPhysicalAlias.11 ==             'rtr-U3:m2:SJ17-2-eng'
entPhysicalAssetID.11 ==           '0007373982'
entPhysicalIsFRU.11 ==             true(1)
```

- Port 1, in Board 3, within Slot 3:

```
entPhysicalDescr.12 ==             'Sun Ethernet-100 Port'
entPhysicalVendorType.12 ==        sun.portTypes.5
entPhysicalContainedIn.12 ==       11
entPhysicalClass.12 ==             port(10)
entPhysicalParentRelPos.12 ==      1
entPhysicalName.12 ==              'P3'
entPhysicalHardwareRev.12 ==       'CC(1.07)'
entPhysicalSoftwareRev.12 ==       '2.0.34'
entPhysicalFirmwareRev.12 ==       '1.1'
entPhysicalSerialNum.12 ==         ''
entPhysicalMfgName.12 ==           'Sun Microsystems'
entPhysicalModelName.12 ==         'SFE-P100'
entPhysicalAlias.12 ==             ''
entPhysicalAssetID.12 ==           ''
entPhysicalIsFRU.12 ==             false(2)
```

- Port 2, in Board 3, within Slot 3:

```
entPhysicalDescr.13 ==            'Sun Ethernet-100 Port'
entPhysicalVendorType.13 ==       sun.portTypes.5
entPhysicalContainedIn.13 ==      11
entPhysicalClass.13 ==            port(10)
entPhysicalParentRelPos.13 ==     2
entPhysicalName.13 ==             'Ethernet B'
entPhysicalHardwareRev.13 ==      'G(1.04)'
entPhysicalSoftwareRev.13 ==      ''
entPhysicalFirmwareRev.13 ==      '1.3'
entPhysicalSerialNum.13 ==        ''
entPhysicalMfgName.13 ==          'Sun Microsystems'
```

EXAMPLE 9–1 Physical Entities for `demo_module_11` *(Continued)*

```
entPhysicalModelName.13 ==        'SFE-P100'
entPhysicalAlias.13 ==            ''
entPhysicalAssetID.13 ==          ''
entPhysicalIsFRU.13 ==            false(2)
```

- Port 3, in Board 3, within Slot 3:

```
entPhysicalDescr.14 ==            'Sun Ethernet-100 Port'
entPhysicalVendorType.14 ==       sun.portTypes.5
entPhysicalContainedIn.14 ==      11
entPhysicalClass.14 ==            port(10)
entPhysicalParentRelPos.14 ==     3
entPhysicalName.14 ==             'Ethernet B'
entPhysicalHardwareRev.14 ==      'G(1.04)'
entPhysicalSoftwareRev.14 ==      ''
entPhysicalFirmwareRev.14 ==      '1.3'
entPhysicalSerialNum.14 ==        ''
entPhysicalMfgName.14 ==          'Sun Microsystems'
entPhysicalModelName.14 ==        'SFE-P100'
entPhysicalAlias.14 ==            ''
entPhysicalAssetID.14 ==          ''
entPhysicalIsFRU.14 ==            false(2)
```

EXAMPLE 9–2 Logical Entities for `demo_module_11`

The `entLogicalTable` is populated with the following values when you run `demo_module_11`:

- Logical Domain "A"

```
entLogicalDescr.1 ==              'Domain A'
entLogicalType.1 ==               solaris
entLogicalCommunity.1 ==          'public-dom1'
entLogicalTAddress.1 ==           124.125.126.127:161
entLogicalTDomain.1 ==            SunExampleDomain
entLogicalContextEngineID.1 ==    ''
entLogicalContextName.1 ==        ''
```

- Logical Domain "B"

```
entLogicalDescr.2 ==              'Domain B'
entLogicalType.2 ==               solaris
entLogicalCommunity.2 ==          'public-dom2'
entLogicalTAddress.2 ==           124.125.126.128:161
entLogicalTDomain.2 ==            SunExampleDomain
entLogicalContextEngineID.2 ==    ''
entLogicalContextName.2 ==        ''
```

EXAMPLE 9–2 Logical Entities for `demo_module_11` *(Continued)*

- Firewall 1

```
entLogicalDescr.3 ==               'Sun Firewall v2.1.1'
entLogicalType.3  ==               dot1dFirewall
entLogicalCommunity.3 ==           'public-firewall1'
entLogicalTAddress.3 ==            124.125.126.129:161
entLogicalTDomain.3 ==             SunExampleDomain
entLogicalContextEngineID.3 ==     ''
entLogicalContextName.3 ==         ''
```

- Firewall 2

```
entLogicalDescr.4 ==               'Sun Firewall v2.1.1'
entLogicalType.4 ==                dot1dFirewall
entLogicalCommunity.4 ==           'public-firewall2'
entLogicalTAddress.4 ==            124.125.126.130:161
entLogicalTDomain.4 ==             SunExampleDomain
entLogicalContextEngineID.4 ==     ''
entLogicalContextName.4 ==         ''
```

Note – `entLogicalTable` does not support SNMPv3 in this example.

EXAMPLE 9–3 Logical to Physical Mappings for `demo_module_11`

The `entLPMappingsTable` is populated with the objects and values in the right column of the following table when you run `demo_module_11`.

Logical Entity and Physical Entity Associations	Logical to Physical Mapping Indexes

Domain A (`entLogicalIndex.1`) uses:
Board 1 (`entPhysicalIndex.5`)
Port 1 (`entPhysicalIndex.12`)
	`entLPPhysicalIndex.1.5  == 5`
	`entLPPhysicalIndex.1.12 == 12`

EXAMPLE 9–3 Logical to Physical Mappings for `demo_module_11` *(Continued)*

Logical Entity and Physical Entity Associations	Logical to Physical Mapping Indexes

Domain B (`entLogicalIndex.2`) uses:
Board 2 (`entPhysicalIndex.8`)
Port 2 (`entPhysicalIndex.13`)
Port 3 (`entPhysicalIndex.14`)
	`entLPPhysicalIndex.2.8 == 8`
	`entLPPhysicalIndex.2.13 == 13`
	`entLPPhysicalIndex.2.14 == 14`

Firewall 1 (`entLogicalIndex.3`) uses:
CPU 1 (`entPhysicalIndex.6`)
Port 1 (`entPhysicalIndex.12`)
	`entLPPhysicalIndex.3.6 == 6`
	`entLPPhysicalIndex.3.12 == 12`

Firewall 2 (`entLogicalIndex.4`) uses:
CPU 3 (`entPhysicalIndex.9`)
Port 2 (`entPhysicalIndex.13`)
Port 3 (`entPhysicalIndex.14`)
	`entLPPhysicalIndex.4.9 == 9`
	`entLPPhysicalIndex.4.13 == 13`
	`entLPPhysicalIndex.4.14 == 14`
	These mappings are included in the `entLPMappingTable` because Firewall 2 uses ports in the board. If the firewall did not use these ports, then a single mapping to the board, for example `entLPPhysicalIndex.4.11` would be sufficient.

EXAMPLE 9–4 Physical to Logical to MIB Alias Mappings for `demo_module_11`

The `entAliasMappingTable` is populated with the following objects and values when you run `demo_module_11`.

If the `ifIndex` values are shared by all logical entities, the `entAliasMappingTable` might be populated as follows:

```
entAliasMappingIdentifier.12.0 ==  ifIndex.1
entAliasMappingIdentifier.13.0 ==  ifIndex.2
entAliasMappingIdentifier.14.0 ==  ifIndex.3
```

The first index in the `entAliasMappingIdentifier` signifies the physical index. In this case, physical entities with the indexes 12, 13, and 14 are Port 1, Port 2, and Port 3. In the preceding `entAliasMappingIdentifier` assignments, Port 1 is mapped to `ifIndex.1`, Port 2 is mapped to `ifIndex.2`, and Port 3 is mapped to `ifIndex.3`. This mapping is for all logical entities that use each of these ports.

If the `ifIndex` values are *not* shared by all logical entities, the `entAliasMappingTable` might be populated as follows:

EXAMPLE 9–4 Physical to Logical to MIB Alias Mappings for `demo_module_11` *(Continued)*

```
entAliasMappingIdentifier.12.0 ==  ifIndex.1
entAliasMappingIdentifier.12.3 ==  ifIndex.101
entAliasMappingIdentifier.13.0 ==  ifIndex.2
entAliasMappingIdentifier.13.3 ==  ifIndex.102
entAliasMappingIdentifier.14.0 ==  ifIndex.3
entAliasMappingIdentifier.14.3 ==  ifIndex.103
```

In this case, one logical entity is mapped differently. Firewall 1, which is `entLogicalIndex.3`, is mapped as follows:

- `ifIndex.101` on Port 1
- `ifIndex.102` on Port 2
- `ifIndex.103` on Port 3

EXAMPLE 9–5 Physical Contains Table Entries for `demo_module_11`

The following table shows the containment relationships among the physical entities. The right column of the table lists the entries added to the `entPhysicalContainsTable` of the Entity MIB by `demo_module_11`.

Physical Entity	Contains	`entPhysicalContainsTable` Entry
Chassis	Slot 1	`entPhysicalChildIndex.1.2 == 2`
	Slot 2	`entPhysicalChildIndex.1.3 == 3`
	Slot 3	`entPhysicalChildIndex.1.4 == 4`
Slot 1	Board 1	`entPhysicalChildIndex.2.5 == 5`
Slot 2	Board 2	`entPhysicalChildIndex.3.8 == 8`
Slot 3	Board 3	`entPhysicalChildIndex.4.11 == 11`
Board 1	CPU 1	`entPhysicalChildIndex.4.6 == 6`
	CPU 2	`entPhysicalChildIndex.4.7 == 7`
Board 2	CPU 3	`entPhysicalChildIndex.8.9 == 9`
	CPU 4	`entPhysicalChildIndex.8.10 == 10`
Board 3	Port 1	`entPhysicalChildIndex.11.12 == 12`
	Port 2	`entPhysicalChildIndex.11.13 == 13`
	Port 3	`entPhysicalChildIndex.11.14 == 14`

CHAPTER 10

Migration of Solstice Enterprise Agents to the System Management Agent

This chapter contains information for developers who want to migrate a subagent from Solstice Enterprise Agents to use in the System Management Agent. The chapter uses `demo_module_12` to illustrate procedures. The following topics are discussed:

Why Migrate to SMA?

Support for the Solstice Enterprise Agents software might be discontinued in a future Solaris release. For this reason, any Solstice Enterprise Agents subagents that you have created must be migrated to use the SMA. In this Solaris release, you can run the Solstice Enterprise Agents software and associated subagents concurrently with the SMA.

The Solstice Enterprise Agents product includes `mibiisa`, a subagent that implements MIB-II and the Sun MIB. In SMA, the functionality of `mibiisa` is implemented by the MIB-II portion of the SMA agent and a new Sun extensions subagent. By default, the `mibiisa` subagent is disabled in this Solaris release.

Requests for MIB-II are handled by the SMA agent directly. Requests for the extensions in the Sun MIB are handled by the `seaExtensions` module, if that module has been loaded. Requests for the Solstice Enterprise Agents master agent, which implements the `snmpdx.mib`, are handled by the `seaProxy` module if that module has been loaded.

The seaProxy module generates dynamic proxies based on static and dynamic Solstice Enterprise Agents subagent registrations. The proxies are not statically defined in snmpd.conf. Note that the seaProxy module does not generate proxies for the mibiisa subagent itself. After the dynamic proxies are generated, the agent's proxy mechanism handles the forwarding of those requests to the Solstice Enterprise Agents master agent.

Solstice Enterprise Agents subagents can still be used with the Solstice Enterprise Agents master agent, and thus with SMA by using the seaProxy module, as explained in *Solaris System Management Agent Administration Guide*. However, SMA support of the Solstice Enterprise Agents software is for a limited transitional time. You should migrate any Solstice Enterprise Agents subagents that you have implemented to use the SMA as early as possible.

Solstice Enterprise Agents Migration Strategy Overview

The general process for implementing a Solstice Enterprise Agents subagent as an SMA module is as follows:

1. Obtain the MIB that was used to create the Solstice Enterprise Agents subagent.
2. Make a copy of the MIB. Name the MIB file according to the guidelines in "MIB File Names" on page 31, if necessary.

   Modify the copy of the MIB for compatibility with mib2c. Use the SUN-SEA-EXTENSIONS-MIB.txt as a model for modifying the MIB. Pay particular attention to the format of the MODULE-IDENTITY group.
3. Use the mib2c tool to generate C code for SMA module templates from the modified MIB.
4. Use the Solstice Enterprise Agents mibcodegen tool to generate C code header and stub files for Solstice Enterprise Agents modules from the original MIB.
5. Compare the template code that mib2c produced to the template code that mibcodegen produced. Examine the instrumentation code from the Solstice Enterprise Agents subagent to determine what you need for instrumentation in the SMA module.
6. Modify the SMA templates to use the appropriate functions to implement similar instrumentation code.

The following section uses an example MIB in demo_module_12 to illustrate this migration process.

Migrating Solstice Enterprise Agent Subagents to SMA

The SMA does not provide a comprehensive tool to migrate a Solstice Enterprise Agents subagent to an SMA module. A Solstice Enterprise Agents subagent uses two types of API functions. One type of API function is used for interaction with the master agent, and the other type is used for custom implementation. The functions for interaction with the master agent are common among all subagents. No tool is available that can separate the two types of functions, and put only the custom implementation code automatically into the corresponding place in the `mib2c`-generated code.

The simplest way to migrate a Solstice Enterprise Agents subagent is first to use the MIB tools of each environment to create code templates for each environment.

The following table compares aspects of the SMA `mib2c` tool and the Solstice Enterprise Agents `mibcodegen` tool. This comparison might help you to understand the code templates that each tool produces.

TABLE 10–1 Comparison of MIB Tools in SMA and Solstice Enterprise Agents Software

	SMA `mib2c` tool	**Solstice Enterprise Agents `mibcodegen` tool**
Scope of action on MIB	`mib2c` is run against individual nodes in a MIB, such as a subtree that contains scalars or a table. Running `mib2c` against individual tables rather than a parent subtree or group is advantageous. You can generate code templates that are customized according to the way you plan to implement each table in SMA. For example, you can generate templates for a table differently if the table is internal or external to the agent.	`mibcodegen` is run against the whole MIB.
Code generated	`mib2c` generates code for the implementation of a module that can be used in SNMP agent or AgentX subagent frameworks. Well-defined APIs are used to expose the functionality.	`mibcodegen` generates code to make the output represent a standalone subagent. SNMP is used to communicate between the master agent and the subagent.

demo_module_12 Code Example for Solstice Enterprise Agents Subagent Migration

The demo_module_12 demonstrates how to implement a Solstice Enterprise Agents subagent as an SMA module.

The demo_module_12 code example is by default located in the directory /usr/demo/sma_snmp/demo_module_12. The README_demo_module_12 file within that directory contains instructions that describe how to perform the following tasks:

- Generate SMA template code from the EXAMPLE-MIB, by running the runmib2c script
- Generate Solstice Enterprise Agents template code from the EXAMPLE-MIB, by running the runmibcodegen script

You should perform the procedures in demo_module_12 to produce the templates that are analyzed in the following section.

Analysis of the demo_module_12 Solstice Enterprise Agents Templates

The mibcodgen tool produced several files. The following table describes and analyzes the files.

TABLE 10–2 Comparison of Solstice Enterprise Agents Templates to SMA Templates

Template File Name	Content	Comparison to SMA Templates
example_tree.c	Contains the type or storage definition for the MIB information.	Only the OID and column definitions contained in this file are also used in templates generated by mib2c. The agent or AgentX frameworks handle the rest for you.
example_stub.h	Contains extern function definitions for all get, set, and free functions that implement the variables in the MIB.	For each SNMP group, mib2c generates an include file that defines externs for similar functions for both scalars and tables.

TABLE 10–2 Comparison of Solstice Enterprise Agents Templates to SMA Templates *(Continued)*

Template File Name	Content	Comparison to SMA Templates
`example_stub.c`	Contains all `get`, `set`, and `free` functions that implement the scalar variables in the MIB.	For each SNMP group, `mib2c` generates a source code file. The file implements code for similar functions for the data types that the group contains, scalars, or tables. `mib2c` also generates the registration code that is invoked at initialization time. The registration code makes the agent aware of the OIDs that are supported. The registration code also identifies the `get` and `set` functions.
`example_rwTableEntry.c`	Contains all `get`, `set`, and `free` functions that implement the column variables for `rwTableEntry` in the MIB.	An equivalent file, `tableType.c` in the example, is generated by `mib2c` with one of the table configuration options. The `mib2c`-generated file contains similar functions but uses very different index handling. `mibcodegen` generates a `get` method that is passed a parameter to indicate whether to perform a `get` or `getnext` request. With `mib2c`, however, the index handling is performed prior to invoking the `get` method to handle a `getnext` request. A `get_first` method is exposed to the SMA agent so that the agent can find the first item in a table. A `get_next` method handles getting the next row in the table. When the correct row is found, the `get` or `set` method is called with the column to manipulate. This process applies to getting the correct row for `get`, `getnext`, or `set` functions when the data is external to the agent. If the data is held by the SMA master agent, table registration involves populating the table. After the table is populated, requests to the table would be handled directly by the SMA master agent.
`example_trap.c`	Contains trap definitions.	`mib2c` does not generate equivalent code. Traps can be generated by calling `send_enterprise_trap_vars()`.

TABLE 10–2 Comparison of Solstice Enterprise Agents Templates to SMA Templates *(Continued)*

Template File Name	Content	Comparison to SMA Templates
`example_appl.c`	Contains code to support subagent.	`mib2c` does not generate equivalent code because such code is not needed. The SMA agent or AgentX framework handles the overhead and invokes the code through API functions.

Modifying the SMA Instrumentation Code

After you generate and analyze the templates, the task then is to extract the core SNMP `get`, `getnext`, and `set` processing out of the Solstice Enterprise Agents subagent code, and move it to the `get` and `set` handler and `get_first/get_next` methods defined in the SMA module approach.

The index handling is removed from each `get` and `set` function in Solstice Enterprise Agents code to be handled by the SMA. Special methods are used for tables. Context fields are used to store the current index information so that advancing in the table is relatively simple.

APPENDIX A

SMA Resources

This appendix lists System Management Agent resources that you might find helpful.

Man Pages

This section lists all the man pages that are associated with the System Management Agent. The man pages are listed in tables, which are organized by the type of content documented in the pages:

- Man Pages for General SNMP Topics
- Man Pages for SNMP Tools
- Man Pages for SNMP Configuration Files
- Man Pages for SNMP Daemons

The following table lists man pages for general SNMP information.

TABLE A–1 Man Pages for General SNMP Topics

Man Page	Description
`sma_snmp`(5)	Gives an overview of the System Management Agent, the Net-SNMP implementation included in the Solaris operating system.
`snmpcmd`(1M)	Describes the common options for Net-SNMP commands.
`snmp_variables`(4)	Discusses the format that must be used to specify variable names to Net-SNMP commands.

The following table lists the man pages for Net-SNMP command tools.

TABLE A–2 Man Pages for SNMP Tools

Man page	Tool Description
`mib2c`(1M)	The `mib2c` tool uses nodes in a MIB definition file to produce two C code template files. The templates can be used as a basis for a MIB module.
`snmpbulkget`(1M)	The `snmpbulkget` utility is an SNMP application that uses the SNMP GETBULK operation to send information to a network manager.
`snmpbulkwalk`(1M)	The `snmpbulkwalk` utility is an SNMP application that uses SNMP GETBULK requests to query a network entity efficiently for a tree of information.
`snmpget`(1M)	The `snmpget` utility is an SNMP application that uses the SNMP GET request to query for information on a network entity.
`snmpgetnext`(1M)	The `snmpgetnext` utility is an SNMP application that uses the SNMP GETNEXT request to query for information on a network entity.
`snmpinform`(1M)	The `snmpinform` command invokes the `snmptrap` utility, which is an SNMP application that uses the SNMP TRAP operation to send information to a network manager.
`snmpnetstat`(1M)	The `snmpnetstat` command symbolically displays the values of various network-related information retrieved from a remote system by using the SNMP protocol.
`snmpset`(1M)	The `snmpset` utility is an SNMP application that uses the SNMP SET request to set information on a network entity.
`snmptrap`(1M)	The `snmptrap` utility is an SNMP application that uses the SNMP TRAP operation to send information to a network manager.
`snmpusm`(1M)	The `snmpusm` utility is an SNMP application that can be used to do simple maintenance on an SNMP agent's User-based Security Module (USM) table.
`snmpvacm`(1M)	The `snmpvacm` utility is a SNMP application that can be used to do maintenance on an SNMP agent's View-based Access Control Module (VACM) table.
`snmpwalk`(1M)	The `snmpwalk` utility is an SNMP application that uses SNMP GETNEXT requests to query a network entity for a tree of information.

TABLE A–2 Man Pages for SNMP Tools *(Continued)*

Man page	Tool Description
`snmpdf`(1M)	The `snmpdf` command is a networked version of the `df` command. `snmpdf` checks the disk space on the remote machine by examining the HOST-RESOURCES-MIB's `hrStorageTable` or the UCD-SNMP-MIB's `dskTable`.
`snmpdelta`(1M)	The `snmpdelta` utility monitors the specified OIDs and reports changes over time.
`snmptable`(1m)	The `snmptable` utility is an SNMP application that repeatedly uses the SNMP GETNEXT or GETBULK requests to query for information on a network entity.
`snmptest`(1M)	The `snmptest` utility is a flexible SNMP application that can monitor and manage information on a network entity. The utility uses a command-line interpreter to enable you to send different types of SNMP requests to target agents.
`snmptranslate`(1m)	The `snmptranslate` utility is an application that translates one or more SNMP object identifier values between symbolic textual forms and numerical forms.
`snmpstatus`(1)	The `snmpstatus` command is an SNMP application that retrieves several important statistics from a network entity.

The following table lists the man pages associated with configuration files that are used by the Net-SNMP agent.

TABLE A–3 Man Pages for SNMP Configuration Files

Man Page	Description
`snmp_config`(4)	Provides an overview of the Net-SNMP configuration files included with System Management Agent.
`snmp.conf`(4)	The file `snmp.conf` defines how the Net-SNMP applications operate. The Net-SNMP applications include `snmpget`, `snmpwalk`, and similar tools.
`snmpd.conf`(4)	The file `snmpd.conf` defines how the Net-SNMP agent operates.
`snmptrapd.conf`(4)	The file `snmptrapd.conf` defines how the Net-SNMP trap-receiving daemon, `snmptrapd`, operates when receiving a trap.

TABLE A–3 Man Pages for SNMP Configuration Files *(Continued)*

Man Page	Description
`snmpconf`(1M)	The `snmpconf` utility is a script that asks you configuration questions. The utility then creates an `snmpd.conf` configuration file that is based on your responses.

The following table lists the man pages for daemons that are associated with Net-SNMP.

TABLE A–4 Man Pages for SNMP Daemons

Man Page	Description
`snmpd`(1M)	The `snmpd` daemon is the SNMP agent. The daemon binds to a port and awaits requests from SNMP management software.
`init.sma`(1M)	The `init.sma` utility is run automatically during installation and system reboot. This utility manages the `snmpd` daemon.
`snmptrapd`(1M)	The `snmptrapd` daemon is an SNMP application that receives and logs SNMP TRAP and INFORM messages.

API Functions

The following Net-SNMP API functions have been tested and are certified to work with the System Management Agent. Documentation from Net-SNMP is provided for all the functions in `/usr/sfw/doc/sma_snmp/html`.

```
netsnmp_mib_handler *netsnmp_create_handler(
                                const char *name,
                                Netsnmp_Node_Handler *handler_access_method);
```

```
netsnmp_handler_registration *netsnmp_create_handler_registration(
                                const char *name,
                                Netsnmp_Node_Handler *handler_access_method,
                                oid * reg_oid,
                                size_t reg_oid_len,
                                int modes);
```

```
void
send_enterprise_trap_vars(int trap,
                          int specific,
                          oid *enterprise,
                          int enterprise_length,
                          netsnmp_variable_list * vars);


void
send_easy_trap(int, int);


void
send_v2trap(netsnmp_variable_list *);


netsnmp_mib_handler *netsnmp_get_debug_handler(void);


void
  netsnmp_init_debug_helper(void);


int
  netsnmp_register_instance(netsnmp_handler_registration *reginfo);


int
  netsnmp_register_read_only_instance(netsnmp_handler_registration *reginfo);


netsnmp_mib_handler *netsnmp_get_instance_handler(void);


netsnmp_mib_handler *netsnmp_get_mode_end_call_handler(
                     netsnmp_mode_handler_list *endlist);


netsnmp_mode_handler_list *netsnmp_mode_end_call_add_mode_callback(
                                 netsnmp_mode_handler_list *endlist,
                                 int mode,
                                 netsnmp_mib_handler *callbackh);


int
  netsnmp_register_scalar(netsnmp_handler_registration *reginfo);


int
  netsnmp_register_read_only_scalar(netsnmp_handler_registration *reginfo);
```

```
netsnmp_mib_handler *netsnmp_get_scalar_handler(void);


netsnmp_mib_handler *netsnmp_get_table_handler(
      netsnmp_table_registration_info


void
  netsnmp_table_helper_add_indexes(va_alist);


int
  netsnmp_register_table_iterator(netsnmp_handler_registration *reginfo,
                                  netsnmp_iterator_info *iinfo);


void *
netsnmp_extract_iterator_context(netsnmp_request_info *);


int
  netsnmp_set_request_error(netsnmp_agent_request_info *reqinfo,
                            netsnmp_request_info *request, int error_value);


int
  snmp_register_callback(int major,
                         int minor,
                         SNMPCallback * new_callback,
                         void *arg);


int
  snmp_call_callbacks(int major,
                      int minor,
                      void *caller_arg);


int
  snmp_unregister_callback(int major,
                           int minor,
                           SNMPCallback * new_callback,
                           void *arg,
                           int matchargs);


void
  snmp_alarm_unregister(unsigned int clientreg);


void
  snmp_alarm_unregister_all(void);
```

```
unsigned int
  snmp_alarm_register(unsigned int when,
                      unsigned int flags,
                      SNMPAlarmCallback * thecallback,
                      void *clientarg);


unsigned int
snmp_alarm_register_hr(struct timeval t,
                       unsigned int flags,
                       SNMPAlarmCallback * cb,
                       void *cd);


int
  snmp_log(int priority, const char *format, ...);


int
  snmp_vlog(int priority, const char *format, va_list ap);


int
  netsnmp_ds_set_boolean(int storeid,
                         int which,
                         int value)


int
  agent_check_and_process(int block)


void
  snmp_shutdown(const char *type)


void
  init_snmp(const char *type)


int
  init_agent(const char *app)


void  *
  netsnmp_request_get_list_data(netsnmp_request_info *request,
  const char *name)


void
  netsnmp_request_add_list_data(netsnmp_request_info *request,
                                netsnmp_data_list *node)


netsnmp_table_request_info *
  netsnmp_extract_table_info(netsnmp_request_info *request)


int
  netsnmp_register_int_instance (const
                        char *name, oid *reg_oid, size_t
                        reg_oid_len, int *it,
                        Netsnmp_Node_Handler *subhandler)


int
  unregister_mib_context (oid *name, size_t len, int priority,
  int range_subid, oid range_ubound, const char *context)


int
  snmp_set_var_typed_value (netsnmp_variable_list *newvar,
  u_char type, const u_char *val_str, size_t val_len)


config_line *
       register_config_handler (const char *type_param,
       const char *token, void(*parser)(const char *, char *),
       void(*releaser)(void), const char *help)


void
  unregister_config_handler (const char
  *type_param, const char *token)


char *
  read_config_read_data (int type, char *readfrom,
                         void *dataptr, size_t *len)


char *
  read_config_store_data (int type, char *storeto, void
                          *dataptr, size_t *len)


netsnmp_delegated_cache *
     netsnmp_create_delegated_cache(
                     netsnmp_mib_handler *handler,
                     netsnmp_handler_registration *reginfo,
                     netsnmp_agent_request_info *reqinfo,
                     netsnmp_request_info *requests,
                     void *localinfo);
```

```
int
snmp_set_var_value (netsnmp_variable_list *var,
                    const u_char *valstr, size_tsize)


void netsnmp_table_set_multi_add_default_row(netsnmp_table_data_set *, ...);

void netsnmp_table_set_multi_add_default_row(va_alist);


netsnmp_table_data_set *netsnmp_create_table_data_set(const char *);
netsnmp_table_set_add_indexes;


int
netsnmp_register_table_data_set(netsnmp_handler_registration *,
                                  netsnmp_table_data_set *,
                                  netsnmp_table_registration_info *);


void send_trap_vars(int trap, int specific, netsnmp_variable_list *vars);
```

APPENDIX **B**

MIBs Implemented in SMA

This appendix lists some of the MIBs that are implemented in the System Management Agent.

MIBs Implemented in SMA

This list includes the MIB modules that have been built into the agent.

UCD-DISKIO-MIB	MIB module for disk IO statistics
RFC1213-MIB	MIB II groups, including IP, TCP, UDP, SYSTEM, ICMP, SNMP, INTERFACES, and STATISTICS. The EGP group is not implemented.
UCD-SNMP-MIB	Memory usage, watch reporting, load averages, virtual memory statistics.
SNMP-USER-BASED-SM-MIB	SNMPv3 user model, security statistics, authentication key information, privacy protocols, USM storage types.
SNMP-VIEW-BASED-ACM-MIB	Group names and access views for View-based Access Control Model (VACM).
UCD-DLMOD-MIB	Names of dynamically loadable modules, location of module, status, dynamic load and unload state.
NET-SNMP-AGENT-MIB	Defines control and monitoring structures for the Net-SNMP agent, gives OIDs and timeout values of all SNMP registered with the agent.

DISMAN-EVENT-MIB	Allows triggering of events and actions for management purposes. The Management Agent for Sun Fire servers uses this MIB.
HOST-RESOURCES-MIB	This MIB is for use in managing host systems. A host in this context is a computer that is used by one or more people. The computer communicates with other similar computers that are attached to the network. The Host Resources MIB does not necessarily apply to devices such as terminal servers, routers, bridges, and monitoring equipment, whose primary function is communications services. However, these types of communication devices are not explicitly precluded from being managed with this MIB. The Host Resources MIB defines attributes that are common to all Internet hosts including, for example, both personal computers and UNIX systems. The MIB also provides Solaris kernel statistics.
SNMP-NOTIFICATION-MIB	The SNMP-NOTIFICATION-MIB module defines MIB objects that provide mechanisms to remotely configure notification parameters. These parameters are used by an SNMP entity for the generation of notifications, or traps.
SUN-SEA-EXTENSIONS-MIB	The SUN-SEA-EXTENSIONS-MIB module describes the Sun-specific extensions to MIB-II.
SUN-SEA-PROXY-MIB	The SUN-SEA-PROXY-MIB is used to manage the Solstice Enterprise Agents `snmpdx` master agent daemon.
AGENTX-MIB	The AGENTX-MIB module is used for the SNMP Agent Extensibility Protocol, AgentX. This MIB module is implemented by the master agent but must be explicitly enabled in order to be used.

Glossary

agent	A software program, typically run on a managed device, that implements the SNMP protocol and services the requests of a manager. Agents can act as proxies for some non-SNMP manageable network nodes.
Agent Extensibility Protocol (AgentX)	A protocol that enables communication between a master SNMP agent and subagents.
ASN.1	Abstract Syntax Notation One. A specification that is used to encode information between a manager and agents in a manner that is independent of the machine and network type.
configuration tokens	Variables that are used for configuring the SNMP agent or modules. Values of tokens can be identifiers, keywords, constants, punctuation, or white space.
context	A collection of managed objects accessible by an SNMP entity. The name for a subset of managed objects.
DAQ	data acquisition. The process of collecting information from a device.
DES	Data Encryption Standard, a standard encryption algorithm used for securing data.
extension	Code that increases the functionality of the SNMP agent. An extension might also be referred to as a MIB module, an extension module, or simply a module.
legacy subagent	A subagent that does not use the AgentX protocol and requires the use of a proxy to communicate with the Net-SNMP agent.
Management Information Base (MIB)	A virtual information store for managed objects. MIBs define the properties of a device that can be managed.
manager	A client application that accesses data from a managed device or system.

master agent	An agent running on a designated SNMP port. The master agent receives SNMP requests from management applications, dispatches the requests to the appropriate subagents, and sends data returned by the subagents to the requester. In addition, the subagents can send traps to the master agent, which are then forwarded to the management application.
MD5	The message digest algorithm, defined in RFC 1321, which converts a message of arbitrary length into a unique 128–bit string. The MD5 algorithm is used to create digital signatures which can be used to verify data integrity.
MIB	Management Information Base.
MIB II	A standard that defines the Management Information Base objects in TCP/IP-based networks that can be managed. MIB II is defined in RFC 1213.
module	Code that increases the functionality of the SNMP agent. A module might also be referred to as a MIB module, an extension module, or an extension.
Net-SNMP	An SNMP agent that is developed as an open source community project. The System Management Agent is based on the Net-SNMP agent.
Network Management Station (NMS)	An application that is used to manage and monitor network devices. The NMS makes SNMP requests to the SNMP agent and receives information from the agent. An NMS is sometimes called a manager or a management application.
Object Identifier (OID)	A sequence of numbers that uniquely identifies each object in a MIB. The OID is a series of integers separated by periods, which indicate the object's place in the MIB tree. For example, the sequence 1.3.6.1.2.1.1.1.0 specifies the system description within the system group of the management subtree.
PDU	Protocol Data Unit. A message, or packet of data, that is transported through network protocol layers. Each layer attaches headers to the packet before passing it along to the next layer. The entire packet, including the user data and headers, is the PDU. SNMP messages consist of a version identifier, an SNMP community name, and a PDU. The PDU types supported in SNMP are GetRequest, GetNextRequest, GetResponse, SetRequest, and Trap.
proxy agent	An agent that acts on behalf of a non-SNMP (foreign) network device. The management station contacts the proxy agent and indicates the identity of the foreign device. The proxy agent translates the protocol interactions it receives from the management station into the interactions supported by the foreign device.

SHA–1	Secure Hash Algorithm - Version 1.0, defined in RFC 3174. SHA is a cryptographic message digest algorithm. The algorithm converts a message into a 160–bit string.
Simple Network Management Protocol (SNMP)	A standard protocol used to manage nodes in the Internet community.
Structure of Management Information (SMI)	An industry-accepted method of organizing object names so that logical access can occur. The SMI states that each managed object must have a name, a syntax, and an encoding. The name, an object identifier (OID), uniquely identifies the object. The syntax defines the data type, such as an integer or a string of octets. The encoding describes how the information associated with the managed objects is serialized for transmission between machines.
subagent	An agent that interacts with a master agent.
trap	A message, sent to a manager, that describes exceptions that occurred on a managed device.
USM	User-based Security Model. A standard for providing SNMP message-level security, described in RFC 3414 at `http://www.ietf.org/rfc/rfc3414.txt`. This RFC document also includes a MIB for remotely monitoring and managing the configuration parameters for the User-based Security Model.
VACM	View-Based Access Control Mechanism A standard for controlling access to management information, described in RFC 3415 at `http://www.ietf.org/rfc/rfc3415.txt`. This RFC document also includes a MIB for remotely managing the configuration parameters for the View-based Access Control Model.

Index