



THE ENTERPRISE MIDDLEWARE SOLUTION

# BEA Manager

## Agent Development Kit Programmer's Guide

Agent Development Kit 4.2  
Document Edition 4.2  
October 1998

# Copyright

Copyright © 1997, 1998 BEA Systems, Inc. All Rights Reserved.

## Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

## Trademarks or Service Marks

BEA, ObjectBroker, TOP END, and TUXEDO are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Connect, BEA Manager, BEA MessageQ, Jolt and M3 are trademarks of BEA Systems, Inc.

All other company names may be trademarks of the respective companies with which they are associated.

## Agent Development Kit Programmer's Guide

| Document Edition | Part Number    | Date         | Software Version          |
|------------------|----------------|--------------|---------------------------|
| 4.2              | 815-001005-001 | October 1998 | Agent Development Kit 4.2 |

---

# Contents

## Preface

|  |      |
|--|------|
| Purpose of This Guide .....              | vii  |
| Who Should Read This Guide .....         | vii  |
| How This Guide Is Organized .....        | viii |
| Online Document Considerations .....     | viii |
| Opening this Guide in a Web Browser..... | ix   |
| Printing from a Web Browser .....        | xi   |
| How to Print the Complete Book .....     | xi   |
| Documentation Conventions .....          | xi   |
| Related Documentation .....              | xiii |
| Agent Development Kit Documentation..... | xiii |
| Additional Documentation .....           | xiv  |
| Other Related Publications .....         | xiv  |
| Contact Information.....                 | xiv  |
| Documentation Support.....               | xiv  |
| Customer Support.....                    | xiv  |

## 1. Getting Started

|   |      |
|---|------|
| Administrative Tasks on UNIX Systems .....                        | 1-1  |
| User Tasks for UNIX Systems .....                                 | 1-4  |
| Agent Startup Tasks on UNIX Systems .....                         | 1-4  |
| Administrative Tasks on Windows NT Systems.....                   | 1-5  |
| User Tasks on Windows NT Systems .....                            | 1-7  |
| Agent Startup Tasks on Windows NT Systems .....                   | 1-7  |
| Executing SNMP Agents Generated by the Agent Development Kit..... | 1-10 |

---

## 2. Agent Development Kit Overview

|  |      |
|--|------|
| Introduction .....                               | 2-1  |
| The Manager/Agent Model .....                    | 2-2  |
| Master Agents and Subagents .....                | 2-3  |
| Managed Objects and Object Identifiers .....     | 2-6  |
| MIB Groups.....                                  | 2-8  |
| Scalar Objects and Tabular Objects .....         | 2-8  |
| Relative and Absolute Object Identifiers .....   | 2-10 |
| Specifying Object Identifiers Symbolically ..... | 2-10 |
| Object Identifier Instance Indexes.....          | 2-11 |
| SNMP Traps .....                                 | 2-12 |
| Anatomy of SNMP Agents.....                      | 2-14 |
| How the Agent Works.....                         | 2-16 |
| Directory Structure .....                        | 2-17 |
| Core Libraries.....                              | 2-18 |

## 3. Tools and Functions

|                                    |      |
|------------------------------------|------|
| Introduction .....                 | 3-1  |
| MIB Variable Definition Files..... | 3-2  |
| SNMP Request Format.....           | 3-2  |
| Agent Tools .....                  | 3-3  |
| build_agent.....                   | 3-4  |
| imibgenall.....                    | 3-6  |
| imibprint.....                     | 3-9  |
| instsrv .....                      | 3-10 |
| snmpget .....                      | 3-11 |
| snmpgetnext .....                  | 3-13 |
| snmpptest.....                     | 3-15 |
| snmptrap .....                     | 3-18 |
| snmptrapd .....                    | 3-21 |
| snmpwalk .....                     | 3-22 |

---

|                              |      |
|------------------------------|------|
| Agent Functions.....         | 3-24 |
| csam_get_keyword.....        | 3-25 |
| csam_set_keyword.....        | 3-27 |
| csam_trap_create.....        | 3-28 |
| csam_trap_add_var_bind ..... | 3-31 |
| csam_trap_send.....          | 3-34 |
| bea_sendtrap.....            | 3-36 |
| bea_snmptrap .....           | 3-38 |

## 4. Developing an SNMP Agent

|   |      |
|---|------|
| Overview of the Development Process .....       | 4-2  |
| Adding Your Custom Code to Generated Files..... | 4-8  |
| Additional Programming Guidelines .....         | 4-12 |
| Community.....                                  | 4-12 |
| Defining Keywords in beamgr.conf.....           | 4-12 |
| Sample Traps Program.....                       | 4-12 |
| Row Deletion .....                              | 4-13 |
| Row Addition.....                               | 4-13 |
| Testing Your Custom Code.....                   | 4-14 |
| Building the Agent Executable.....              | 4-15 |
| Object Identifier Lists.....                    | 4-16 |
| Sample sorted_oid_list.....                     | 4-17 |
| Sample ascii_oid_list .....                     | 4-17 |
| Testing the Agent .....                         | 4-18 |
| Installing the New Agent.....                   | 4-19 |
| Troubleshooting .....                           | 4-19 |
| MIB Modification.....                           | 4-20 |
| Using Multiple SNMP Agents.....                 | 4-21 |

---

## 5. Generated Access Function Templates

|                                 |      |
|---------------------------------|------|
| Function Templates .....        | 5-2  |
| init_MIBRootName .....          | 5-2  |
| refresh_MIBRootName .....       | 5-2  |
| test_ObjectName .....           | 5-4  |
| set_ObjectName .....            | 5-5  |
| test_TableName_row_create ..... | 5-6  |
| set_TableName_row_create .....  | 5-7  |
| Constants and Variables .....   | 5-8  |
| DELTA_TableName_ENTRIES .....   | 5-8  |
| INIT_TableName_ENTRIES .....    | 5-8  |
| MAX_StringObjectName .....      | 5-8  |
| MIBRootName_refresh_rate .....  | 5-9  |
| v_MIBRootName .....             | 5-9  |
| Generated Functions .....       | 5-11 |
| get_ObjectName .....            | 5-11 |

## Index

---

# Preface

## Purpose of This Guide

This *Agent Development Kit Programmer's Guide* provides a detailed explanation of how to create your own SNMP agent or SMUX subagent to support your private Management Information Base (MIB). A set of application programming interfaces (APIs) are also provided for generating SNMP traps.

Use this manual to understand:

- ◆ How to build your own SNMP agent or SMUX subagent
- ◆ The Agent Development Kit APIs
- ◆ Source code components built for you by the Agent Development Kit code generator/MIB compiler
- ◆ The Agent Development Kit directory structure

## Who Should Read This Guide

This guide is particularly helpful for programmers responsible for building an SNMP agent or SMUX subagent to support a private MIB.

---

# How This Guide Is Organized

The *Agent Development Kit Programmer's Guide* is organized as follows:

- ◆ Chapter 1, “Getting Started,” explains the post-installation steps to prepare for using the Agent Development Kit.
- ◆ Chapter 2, “Agent Development Kit Overview,” explains the agent/manager model, the structure of the SNMP agents built using the Agent Development Kit, and the Agent Development Kit directory structure.
- ◆ Chapter 3, “Tools and Functions,” describes the tools and APIs provided for building and testing an SNMP agent.
- ◆ Chapter 4, “Developing an SNMP Agent,” explains the steps you need to build an agent to support your private MIB.
- ◆ Chapter 5, “Generated Access Function Templates,” describes the access function skeletons and other source code components generated for you by the Agent Development Kit code generator/MIB compiler.

## Online Document Considerations

This book, *Agent Development Kit Programmer's Guide*, is designed primarily as an online, hypertext guide. If you are reading this as a paper publication, note that to get full use from this guide you should install and access it as an online document via a Web browser that supports HTML 3.0. Netscape Navigator 2.02 or Microsoft Internet Explorer 3.0 or later are recommended. (Information on how to install the online documentation is available in the *BEA Manager Release Notes*.



## Opening this Guide in a Web Browser

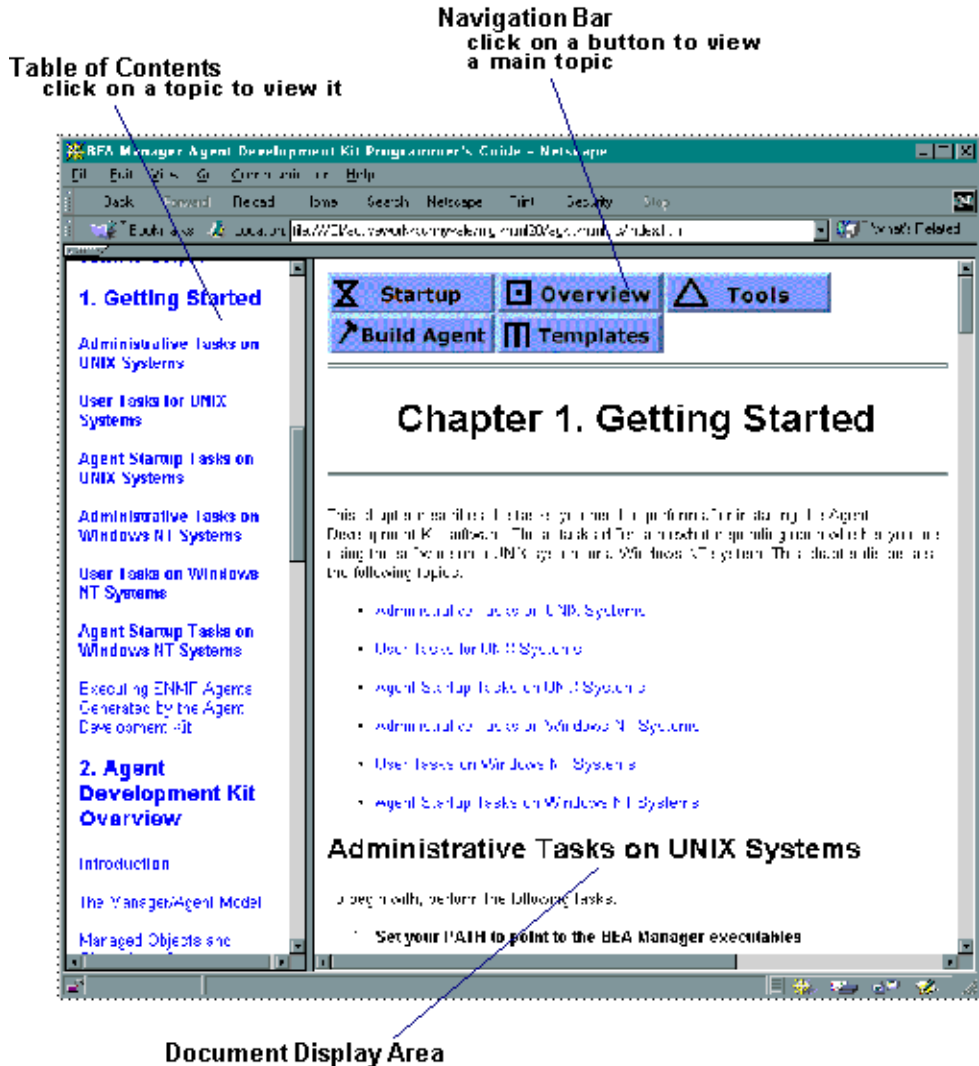
To access the interactive version of this document, open the following HTML file in a Web browser:

`$RELEASE_ROOT/DOCS/MGR/20/AGCON/INDEX.HTM`

**Note:** The online documentation requires a Web browser that supports HTML 3.0. Netscape Navigator 2.02 or Microsoft Internet Explorer 3.0 or later are recommended.

Figure 1 shows the online guide with the clickable navigation bar and table of contents.

**Figure 1 Agent Development Kit Programmer's Guide Displayed in Netscape Web Browser**



## Printing from a Web Browser

You can print a hardcopy version of this document, one file at a time, from the Web browser. Before you print, make sure that the chapter or appendix you want is displayed and *selected* in your browser. (To select a file, click anywhere inside the frame you want to print. If your browser offers a Print Preview feature, you can use it to verify which file you are about to print.)

## How to Print the Complete Book

A PDF version of this online help is available in the following location:

`$RELEASE_ROOT/DOCS/MGR/20/PDF/AGKIT.PDF`

To print the documentation, open a PDF file in an Adobe Acrobat Reader and choose the file print option.

If you do not have a reader, you can download one from the Adobe Web site at <http://www.adobe.com/>.

## Documentation Conventions

The following documentation conventions are used throughout this manual.

| Convention           | Item   |
|----------------------|--|
| <b>boldface text</b> | Indicates terms defined in the glossary.                     |
| Ctrl+Tab             | Indicates that you must press two or more keys sequentially. |
| <i>italics</i>       | Indicates emphasis or book titles.                           |

---

| Convention  | Item  |
|---|---|
| <code>monospace text</code>                         | Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.<br><i>Examples:</i><br><code>#include &lt;iostream.h&gt; void main ( ) the pointer psz<br/>chmod u+w *<br/>\tux\data\ap<br/>.doc<br/>tux.doc<br/>BITMAP<br/>float</code> |
| <b><code>monospace<br/>boldface<br/>text</code></b> | Identifies significant words in code.<br><i>Example:</i><br><code>void <b>commit</b> ( )</code>   |
| <i><code>monospace<br/>italic<br/>text</code></i>   | Identifies variables in code.<br><i>Example:</i><br><code>String <i>expr</i></code>   |
| UPPERCASE<br>TEXT                                   | Indicates device names, environment variables, and logical operators.<br><i>Examples:</i><br><code>LPT1<br/>SIGNON<br/>OR</code>  |
| <code>{ }</code>                                    | Indicates a set of choices in a syntax line. The braces themselves should never be typed.   |
| <code>[ ]</code>                                    | Indicates optional items in a syntax line. The brackets themselves should never be typed.<br><i>Example:</i><br><code>buildobjclient [-v] [-o name ] [-f <i>file-list</i>]...<br/>[-l <i>file-list</i>]...</code>   |
| <code> </code>                                      | Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.   |

| Convention | Item   |
|------------|--|
| ...        | Indicates one of the following in a command line: <ul style="list-style-type: none"><li>◆ That an argument can be repeated several times in a command line</li><li>◆ That the statement omits additional optional arguments</li><li>◆ That you can enter additional parameters, values, or other information</li></ul> The ellipsis itself should never be typed. <p><i>Example:</i></p> <pre>buildobjclient [-v] [-o name ] [-f file-list]...<br/>[-l file-list]...</pre> |
| .          | Indicates the omission of items from a code example or from a syntax line.   |
| .          | The vertical ellipsis itself should never be typed.  |
| .          |  |

## Related Documentation

The following sections list the documentation provided with the Agent Development Kit, and other publications related to BEA Manager technology.

### Agent Development Kit Documentation

The Agent Development Kit documentation consists of the following items:

- ◆ *Agent Development Kit Programmer's Guide*
- ◆ *BEA Manager Installation Guide*
- ◆ *BEA Manager Release Notes*

---

## Additional Documentation

Readers of this guide should also consult the following BEA Manager documentation:

- ◆ *Agent Integrator Reference Manual*

## Other Related Publications

A comprehensive list of relevant publications can be found in Appendix A, “SNMP Information,” in the *Agent Integrator Reference Manual*.

## Contact Information

The following sections provide information about how to obtain support for the documentation and software.

## Documentation Support

If you have questions or comments on the documentation, you can contact the BEA Information Engineering Group by e-mail at **docsupport@beasys.com**. (For information on how to contact Customer Support, refer to the section “Customer Support.”)

## Customer Support

If you have any questions about this version of BEA Agent Development Kit, or if you have problems installing and running BEA Agent Development Kit, contact BEA Customer Support through BEA WebSupport at [www.beasys.com](http://www.beasys.com). You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- ◆ Your name, e-mail address, phone number, and fax number
- ◆ Your company name and company address
- ◆ Your machine type and authorization codes
- ◆ The name and version of the product you are using
- ◆ A description of the problem and the content of pertinent error messages





# 1 Getting Started

This chapter describes the tasks you need to perform after installing the Agent Development Kit software. These tasks differ somewhat depending upon whether you are using the software on a UNIX system or a Windows NT system. This chapter discusses the following topics:

- ◆ Administrative Tasks on UNIX Systems
- ◆ User Tasks for UNIX Systems
- ◆ Agent Startup Tasks on UNIX Systems
- ◆ Administrative Tasks on Windows NT Systems
- ◆ User Tasks on Windows NT Systems
- ◆ Agent Startup Tasks on Windows NT Systems

## Administrative Tasks on UNIX Systems

To begin with, perform the following tasks:

### 1. Set your **PATH** to point to the BEA Manager executables

All users of the installed BEA Manager products will need to update their **PATH** environment variable to include the location of the BEA Manager executable files. The following is a C shell example:

```
% set path = ( $PATH installation_directory/bin )
```

## 2. Copy the configuration file

Log in as root and copy the BEA Manager configuration file `beamgr.conf` from `installation_directory/etc` to the `/etc` directory.

```
%su
Password:
# cp installation_directory/etc/beamgr.conf /etc
```

## 3. Specify the destination for traps, if desired

The default destination for SNMP trap notifications is `localhost`. If you want to set a different target destination for SNMP traps, use your favorite text editor to modify the BEA Manager configuration file (`beamgr.conf`) `TRAP_HOST` entry. The `TRAP_HOST` entry is used to specify the host name of the target destination machine for SNMP trap notifications, and the port number and community name to use in sending traps. For more information refer to the “Configuration Files” chapter in the *Agent Integrator Reference Manual*.

## 4. Specify non-default SNMP communities and SMUX password, if desired

By default, BEA Manager agents (such as the Agent Integrator or `tux_snmpd` when running as an SNMP agent) use `public` as the read-only community and `iview` as the read-write community when communicating with SNMP managers. If you want to specify different community names to be used by BEA Manager SNMP agents, this is specified in the BEA Manager passwords file. The passwords file can also be used to specify a password to be used by Agent Integrator for authenticating connection requests from SMUX subagents. To set up the passwords file, do the following:

- a. Copy the BEA Manager passwords file (`beamgr_snmpd.conf`) from the `installation_directory/etc` to the `/etc` directory and make the copy readable and writable only by root. For example:

```
# cp installation_directory/etc/beamgr_snmpd.conf /etc
# chmod 600 /etc/beamgr_snmpd.conf
```

- b. Now you can edit the copied file to update your SNMP communities. The keywords in this file are:

- ◆ `SMUX_PASSWD`
- ◆ `COMMUNITY_RO`
- ◆ `COMMUNITY_RW`

For more information refer to the “Configuration Files” chapter in the *Agent Integrator Reference Manual*.

## 5. Advertise services if you need to use non-standard ports

By default, BEA Manager agents assume the following port numbers as specified by SNMP and SMUX standards:

|           |         |
|-----------|---------|
| snmp      | 161/udp |
| snmp-trap | 162/udp |
| smux      | 199/tcp |

The default port assignments may be sufficient for your needs. If necessary, you can define these services on other ports, or use the appropriate command-line options when starting BEA Manager agents to assign them to non-default ports.

- a. To modify or define the services, determine if the NIS is running. You can use the `ypwhich` command to determine if an NIS server or map master is available. For example:

```
% ypwhich
zort.kremvax.com
```

- b. If an NIS server is available, you can use the `ypcat` command to determine if the services are available.

```
% ypcat services | grep snmp
snmp-trap      162/udp      snmptrap
snmp           161/udp
```

- c. If an NIS server is not available and services are provided on the local host, you can examine the `/etc/services` file instead.

```
% cat /etc/services | grep snmp
snmp-trap      162/udp      snmptrap
snmp           161/udp
```

Refer to your UNIX system documentation, or consult your UNIX system administrator, for instructions specific to your UNIX platform to establish the SNMP services if necessary.

# User Tasks for UNIX Systems

All users of the Agent Development Kit have to set certain environment variables for the product to function properly. In addition to the environment variables shared with other BEA Manager products, the following environment variable needs to be set:

**RELEASE\_ROOT**

Must contain the location of the installed files.

The following example uses C shell commands:

```
% setenv RELEASE_ROOT /net/tuxedo
```

## Agent Startup Tasks on UNIX Systems

Before running the Agent Integrator or other agents supplied with the Agent Development Kit, or an agent built using the Agent Development Kit, perform the following tasks:

### **1. Modify the BEA Manager configuration file entries, if desired**

If you are using the `unix_snmpd` subagent, you may want to modify the following fields in the BEA Manager configuration file (`beamgr.conf`) after copying it to `/etc`:

- ◆ `SYS_DESCR`
- ◆ `SYS_INSTALL`
- ◆ `SYS_CONTACT`
- ◆ `SYS_NAME`
- ◆ `SYS_LOCATION`
- ◆ `SYS_SERVICES`

These entries correspond to MIB objects in the `beaSystem` MIB group supported by the `unix_snmpd` subagent.

Refer to the “Configuration Files” chapter in the *Agent Integrator Reference Manual* for details on these entries.

## 2. Start the agents

Now, log in as root and start the agents in the following order:

```
% su
Password:
# snmp_integrator
# unix_snmpd
```

# Administrative Tasks on Windows NT Systems

After installing the Agent Development Kit software, perform the following tasks:

## 1. Install the BEA Manager configuration file

Copy the BEA Manager configuration file (`beamgr.conf`):

```
md c:\etc
copy installation-directory\etc\beamgr.conf c:\etc
```

## 2. Specify the destination for traps, if desired

The default destination for SNMP trap notifications is `localhost`. If you want to set a different target destination for SNMP traps, use your favorite text editor to modify the BEA Manager configuration file (`beamgr.conf`) `TRAP_HOST` entry. The `TRAP_HOST` entry is used to specify the host name of the target destination machine for SNMP trap notifications, and the port number and community name to use in sending traps. For more information refer to the “Configuration Files” chapter in the *Agent Integrator Reference Manual*.

## 3. Specify non-default SNMP communities and SMUX password (if desired)

By default, BEA Manager agents (such as the Agent Integrator or `tux_snmpd` when running as an SNMP agent) use `public` as the read-only community and `iview` as the write-read community when communicating with SNMP managers. If you want to specify different community names to be used by BEA Manager SNMP agents, this is specified in the BEA Manager passwords file. The passwords file can also be used to specify a password to be used by Agent

Integrator for authenticating connection requests from SMUX subagents. To set up the passwords file, do the following:

- a. Copy the BEA Manager passwords file (`beamgr_snmpd.conf`) to `c:\etc`. For example:

```
copy installation-directory\etc\beamgr_snmpd.conf c:\etc
```

- b. Now you can modify the SNMP communities in this file. The keywords used in this file are:

- ◆ `SMUX_PASSWD`
- ◆ `COMMUNITY_RO`
- ◆ `COMMUNITY_RW`

For more information refer to the “Configuration Files” chapter in the *Agent Integrator Reference Manual*.

#### 4. Advertise services if you need to use non-standard ports

By default, BEA Manager agents assume the following port numbers as specified by SNMP and SMUX standards:

|                        |                      |
|------------------------|----------------------|
| <code>snmp</code>      | <code>161/udp</code> |
| <code>snmp-trap</code> | <code>162/udp</code> |
| <code>smux</code>      | <code>199/tcp</code> |

The default port assignments may be sufficient for your needs. If necessary, you can define these services on other ports, or use the appropriate command-line options when starting BEA Manager agents to assign them to non-default ports.

To modify or define the service locally, add the appropriate lines in the `NT-root-directory\system32\drivers\etc\services` file. You may wish to consult your system administrator.

# User Tasks on Windows NT Systems

All users of the Agent Development Kit have to set certain environment variables for the product to function properly.

## **RELEASE\_ROOT**

Must contain the location of the installed files.

## **PATH**

Should include the location of the Agent Development Kit executable files.

For example:

```
set RELEASE_ROOT=c:\net\tuxedo
set PATH=%RELEASE_ROOT%\bin;%PATH%
```

# Agent Startup Tasks on Windows NT Systems

Before running the Agent Integrator or other agents supplied with the Agent Development Kit, or any agents built using the Agent Development Kit, carry out the following tasks:

## **1. Modify entries in the configuration file (if desired)**

Before running any agents built with the Agent Development Kit, you may need to modify the following fields in the BEA Manager configuration file (`beamgr.conf`):

- ◆ **SYS\_DESCR**
- ◆ **SYS\_INSTALL**
- ◆ **SYS\_CONTACT**
- ◆ **SYS\_NAME**
- ◆ **SYS\_LOCATION**
- ◆ **SYS\_SERVICES**

These entries correspond to MIB objects in the `beaSystem` MIB group supported by the `nt_snmpd` subagent.

Refer to the “Configuration Files” chapter of the *Agent Integrator Reference Manual* for details on these entries.

## 2. Start the agents from the Windows NT service panel

The Agent Integrator and agents are installed as Windows NT services.

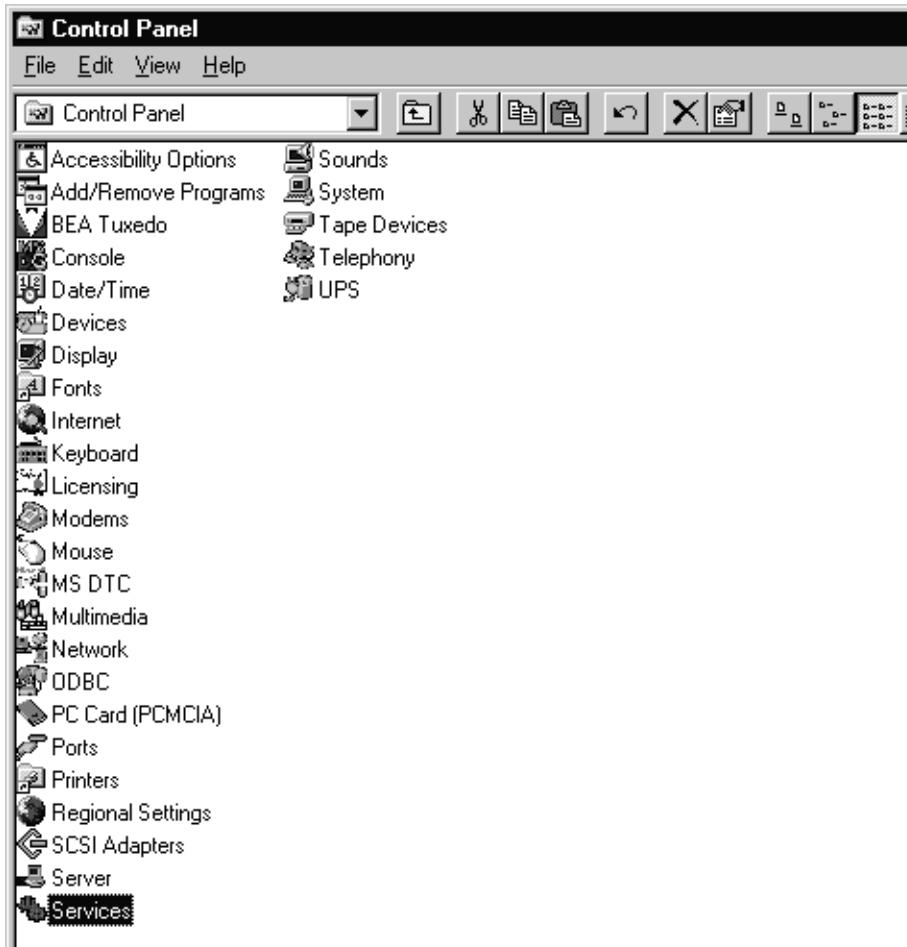
- a. The Agent Integrator is installed as a Windows NT service (`snmp_integrator`).
- b. A system agent is installed as a Windows NT service (`nt_snmpd`).

For starting agents generated using the Agent Development Kit, refer to the next section, “Executing SNMP Agents Generated by the Agent Development Kit.”

The Integrator and agents must be started from the Services control panel. Expose the Windows Taskbar and select the **Start** Button. Select Settings from the menu, and open the Control Panel. Double-click on the Services applet. From the Taskbar, select Start→Settings→Control Panel→Services.

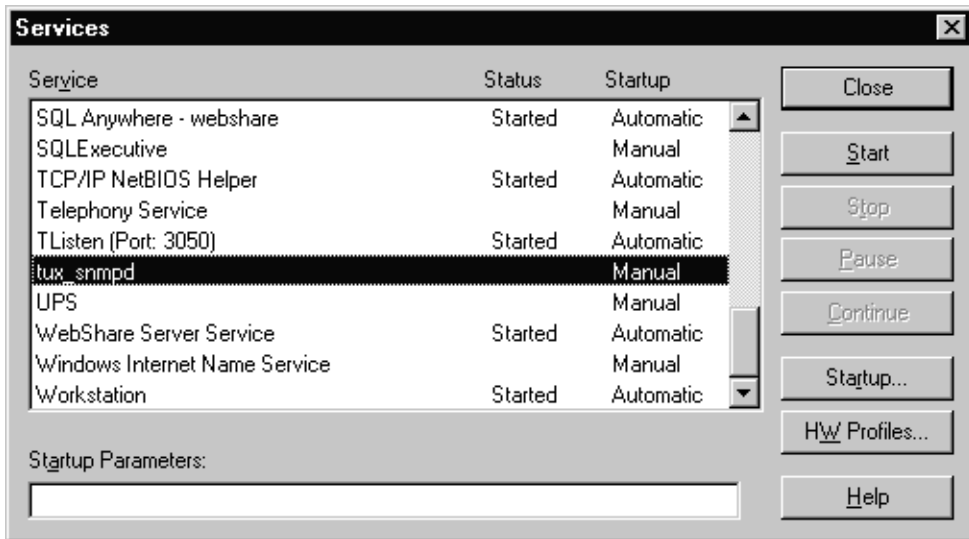
Locate each of the installed services. Agent Integrator (`snmp_integrator`) must be started prior to the subagents (but *after* any `NON_SMUX_PEERS`).



**Figure 1-1** Selecting Services from the Control Panel

Locate the installed service (`tux_snmpd`) and select Start to start it. There may be a short delay as the service is initiated.

**Figure 1-2 Starting a Selected Service**



If a SMUX master agent (e.g., `snmp_integrator`) is not running, the user must provide `-s` as a start-up parameter before selecting **Start**.

## Executing SNMP Agents Generated by the Agent Development Kit

On a Windows NT system, the SNMP agents generated by the Agent Development Kit need to be installed as a service. To install an agent as a service (e.g., to install `my_snmpd.exe`), enter the command:

```
instsrv.exe my_snmpd absolute-path\my_snmpd.exe
```

Once this command is executed, start the `my_snmpd` service from the Service Control Panel. From the Taskbar, select Start→Settings→Control Panel→Services.

# 2 Agent Development Kit Overview

This chapter discusses the following topics:

- ◆ “Introduction,” lists recommended reading, and gives a brief overview of the manager/agent model, the concept of a managed object, object identifiers, and SNMP traps.
- ◆ “Anatomy of SNMP Agents,” describes the components of SNMP agents built using the Agent Development Kit: agent core, agent hooks, and the access functions.
- ◆ “Directory Structure,” describes the Agent Development Kit directory structure.

## Introduction

This manual explains how to:

- ◆ Create an agent that uses Simple Network Management Protocol (SNMP) to enable management of a resource defined in a private management information base (MIB).
- ◆ Create a subagent that provides manageability of a resource by plugging into an SNMP master agent using SNMP Multiplex (SMUX) protocol.

The Agent Development Kit simplifies this work by providing automated code generation and additional features that allow you to build an agent quickly and reliably.

You do not need to understand the SNMP or SMUX protocol in depth to build an SNMP agent with this Agent Development Kit. We do recommend that you read the RFC documents listed below.

A necessary step in building an agent or subagent is defining the manageable features of the resource in an SNMP-compliant MIB. To do this, you should have some knowledge of Abstract Syntax Notation One (ASN.1), as defined in the International Organization for Standardization (ISO) standards. This product also presupposes that the user is familiar with the C programming language.

The Agent Development Kit is compliant with the following Internet standards:

RFC 1155 — Structure of Management Information

RFC 1157 — Simple Network Management Protocol

RFC 1212 — Concise MIB Definitions

RFC 1213 — Management Information Base (MIB II)

RFC 1227 — SNMP Multiplex (SMUX)

Information on obtaining these RFC documents can be found in the “SNMP Information” appendix in the *Agent Integrator Reference Manual*.

## The Manager/Agent Model

The Agent Development Kit software is based on the manager/agent model described in the ISO network management standards. In this model, a network manager exchanges monitoring and control information about network and system resources with distributed software processes called *agents*.

Any system or network resource that is manageable through this exchange of information is a *managed resource*. This could be a software resource, such as a message queue or TUXEDO application, or a hardware resource such as a router or NFS file server.

Agents function as “collection devices” that typically gather and send data about the managed resource in response to a request from a manager. In addition, agents often have the ability to issue unsolicited reports to managers when they detect, or “trap,” certain predefined thresholds or events on a managed resource. In SNMP terminology, these unsolicited event reports are called *trap notifications*.

A manager relies upon a database of definitions and information about the properties of managed resources and the services the agents support — this comprises the management information base (MIB). When new agents are added to extend the management domain of a manager, the manager must be provided with a new MIB component that defines the manageable features of the resources managed through that agent. The manageable features of resources, as defined in an SNMP-compliant MIB, are called *managed objects*. When the heterogeneous components of an enterprise's distributed systems are defined within a common MIB on the management station, this provides a unified perspective and single access point for managing systems made up of diverse hardware and software components from numerous vendors.

In order to provide manageability of a resource, an agent must have access to *instrumentation* in the managed resource. This consists of some means of obtaining current values of the manageable features of that resource, or a means of modifying those values. This might consist of operating system calls, or an application's application programming interface (API) that provides status and performance information about that application.

## Master Agents and Subagents

In the SNMP architecture, each managed object can be managed through only one SNMP agent, and each host may have only one agent communicating with an SNMP manager. The original SNMP management solution allowed for only a single monolithic agent to carry out all management responsibilities on a given network node (IP address). This solution was soon discovered to be too inflexible to provide for effective management of increasingly complex systems. In addition to the agents typically provided by computer manufacturers for hardware and operating system information, agents are also being produced by vendors of other products, such as agents for SQL database systems. Complex and heterogeneous systems thus require the ability to accommodate multiple agents on a single network node.

The SNMP architecture was thus extended to allow a single master agent to communicate with subagents, allowing multiple agents to cooperate in managing diverse hardware and software components on a single host. This master agent functionality is provided by the BEA Manager Agent Integrator software. The extended SNMP Manager/Agent model is illustrated in Figure 2-1.

One of the most popular protocols used for communication between an SNMP master agent and subagents is the SNMP Multiplex (SMUX) protocol, defined in RFC 1227.

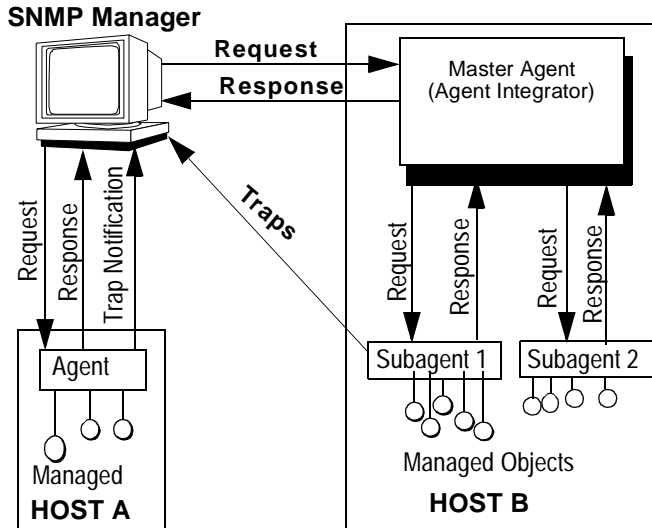
However, no standardized solution has emerged for coordination of multiple non-SMUX SNMP agents on a single host. Master agents that “speak” SMUX protocol to subagents are typically able to communicate only with SMUX-compliant subagents, and cannot communicate with non-SMUX SNMP agents running on the same host.

The BEA Agent Integrator is an intelligent master agent that provides a comprehensive solution for orchestration of diverse agents and subagents.

The Agent Integrator is a master agent that can run on the same node with monolithic SNMP agents and SMUX subagents, allowing multiple agents and subagents from any vendor to cooperate in the management of system components. The multiple SNMP agents and SMUX subagents communicate with SNMP managers through the Integrator and appear as a single SNMP agent to any SNMP manager.

In its communication with SMUX subagents (and non-SMUX SNMP agents running on the same host), the master agent acts as a proxy for the manager. The master agent distributes requests from the manager to specific SNMP agents or subagents, and receives the responses from the individual agents and forwards those responses back to the manager (as illustrated in Figure 2-1).

Figure 2-1 SNMP Architecture



Refer to the *Agent Integrator Reference Manual* for more information on the Agent Integrator.

The Agent Development Kit is a software development kit for the creation of SMUX subagents or SNMP agents for management of system resources, as defined in private MIBs. Subagents created using the Agent Development Kit are also capable of running as a stand-alone SNMP agent. The Agent Development Kit is fully compliant with RFC 1212, Concise MIB Definitions.

## Managed Objects and Object Identifiers

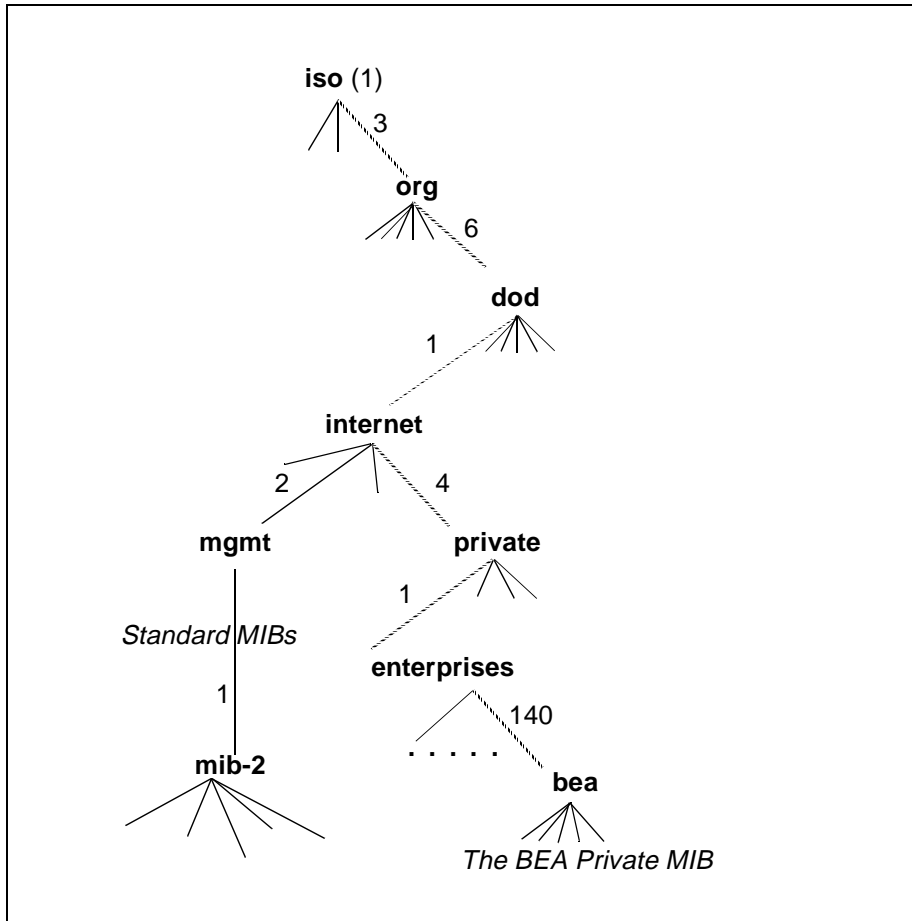
The manager sees the data it manages as a collection of managed objects, represented in ASN.1 notation. The ASN.1 notation defines the data that the manager and the agent exchange. ASN.1 is a formal language described in the ISO 8824 and 8825 standards.

The managed objects comprise the management information base (MIB). Each object in the MIB has an *object identifier* (OID), which the manager uses to request its value from the agent. The OID is a sequence of integers that uniquely identify a managed object by defining a path to that object through a tree-like structure, which is often called the *OID tree* or registration tree. When an SNMP agent wishes to access a specific object, it traverses the OID tree to find the object.

The following diagram illustrates the OID tree for the BEA private MIB. Each BEA private MIB object that the SNMP agent software manages has a unique object identifier, using a prefix of .1.3.6.1.4.1.140 to identify it as an object in the BEA private MIB. This number sequence defines the path to this branch of the OID tree, represented by dashed lines in Figure 2-2.



Figure 2-2 Object Identification Scheme

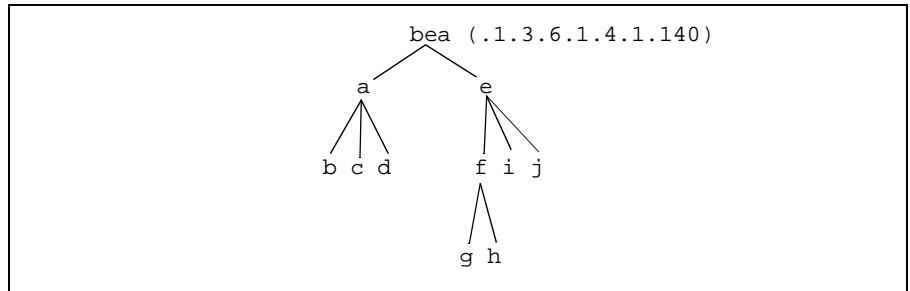


All standard MIBs reside under `mgmt (.1.3.6.1.2)` in this diagram — for example, MIB II (`.1.3.6.1.2.1`). The distinction between the standard and private MIBs is that of control over the object definitions. Standard MIBs are those that have been approved by the Internet Activities Board (IAB). MIBs defined unilaterally by equipment and software vendors are initially defined as private MIBs under `private.enterprises`. A branch within the `private.enterprises` subtree is allocated to each vendor that registers for an enterprises object identifier. For a complete listing of objects in the BEA private MIB in ASN.1 notation, read the file `bea.asn1` in the Agent Development Kit distribution.

## MIB Groups

A MIB *group* is a collection of managed objects, and is itself represented by the name or OID of a node in the OID tree. Groups may contain other groups. For example, `bea` is a MIB group that is a member of the `private.enterprises` MIB group.

The nodes in the OID tree that are not groups — the base level of the OID tree — are the “leaves” of the OID tree. For example, in the following diagram:



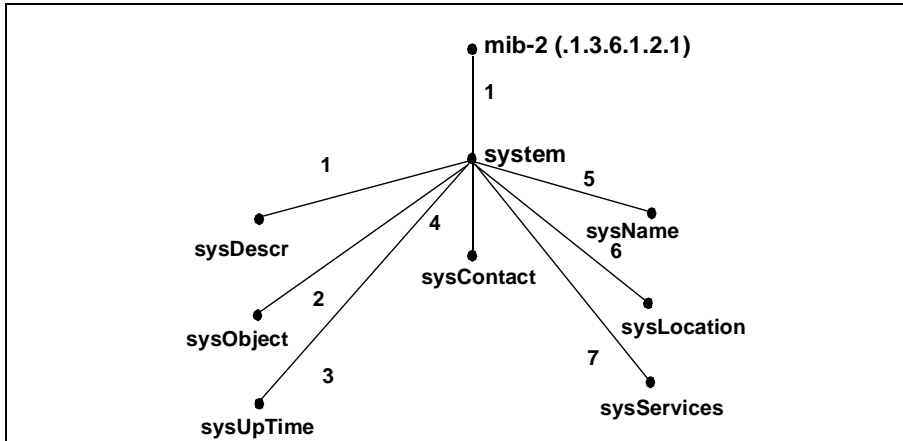
- ◆ MIB group `a` contains a group of objects, `b`, `c`, and `d`
- ◆ MIB group `e` contains a group of objects, `f`, `i`, and `j`
- ◆ MIB group `f` contains a group of objects, `g` and `h`

## Scalar Objects and Tabular Objects

A managed object has both a type (defined in ASN.1) and a value. For example, the SNMP system group object `sysLocation` for the node `blueberry` has the type `DisplayString` and might have the value “Server Room 1242, Bldg. 11.”

Managed objects, in SNMP, are of two types: scalar objects and tabular objects. A managed object that always has a single instance is called a *scalar object*. *Tabular objects*, on the other hand, have multiple instances, such as the rows of a table. For example, the MIB II `system` group has seven “leaf” objects under it, as illustrated in Figure 2-3. Each of these objects is a scalar object. For example, the value of `sysUpTime` is the length of time since re-initialization of a system’s network management software (SNMP agent), measured in hundredths of a second.

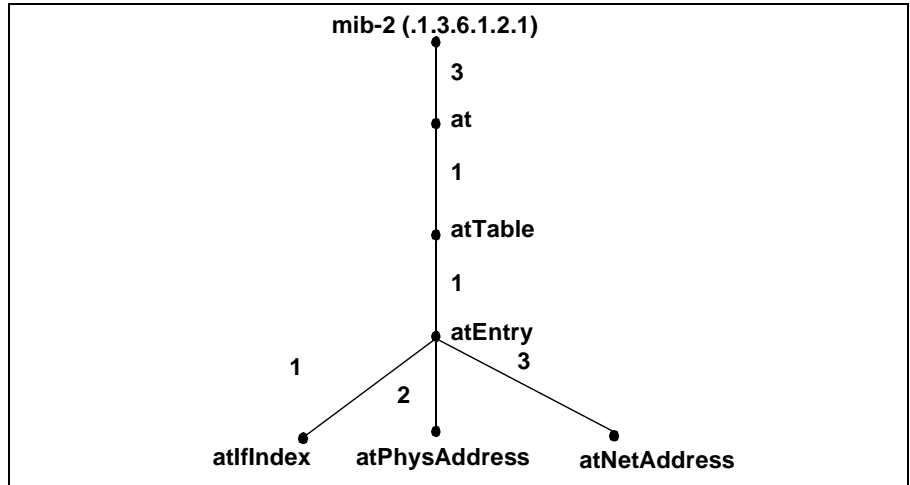
Figure 2-3 A Group of Scalar Objects



Tables in SNMP are two-dimensional objects defined as an ASN.1 type called **SEQUENCE OF**, which allows 0 or more members. Each element of the sequence is an entry (row) in the table, which is itself a sequence of scalar-valued objects. SNMP does not allow tables to be nested within tables.

For example, the MIB II *at* group (illustrated in Figure 2-4) contains simply one tabular object, the *atTable*, which contains one row for each of a system's physical interfaces. Each row in the table is an instance of the object *atEntry*. Each row contains instances of the scalar-valued leaf objects *atIfIndex*, *atPhysAddress*, and *atNetAddress*. The leaf objects are called *columnar objects* since the instances of each such object constitute one column in the table. Although these objects have scalar-valued instances, they are not scalar objects because they can have multiple instances.

Figure 2-4 A Group Containing a Tabular Object



## Relative and Absolute Object Identifiers

The examples of object identifiers discussed thus far all specify a path to the attribute from the root of the OID tree. For example, the BEA object identifier .1.3.6.1.4.1.140 specifies the path from the root of the tree. (The root does not have a name or a number but the initial 1 in this OID is directly below root.) These are called *absolute* OIDs. However, a path to the attribute may be specified relative to some node in the OID tree. For example, 2.1.1.7 specifies the sysContact object in the system group, relative to the internet node in the OID tree. This is called a *relative* OID.

## Specifying Object Identifiers Symbolically

Until now we have used only a series of integers separated by dots — called “dot-dot” notation — to describe OIDs. However, an OID can also be expressed symbolically, or by a combination of both integers and textual symbols. A symbolic OID uses mnemonic keywords to specify the managed object; for example:

```
mgmt.mib.system.sysDescr
```

The following numeric OID uses integers to specify the same managed object:

```
2.1.1.1
```

Note that this example is a relative OID.

An OID may combine both symbolic and numeric representations of individual nodes of the OID tree; for example:

```
mgmt.mib.1.sysDescr
```

Absolute OID names always begin with a dot and must specify every node of the OID tree from the top-most node to the specific managed object:

```
.iso.org.dod.internet.mgmt.mib.system.sysDescr
```

```
.1.3.6.1.2.1.1.1
```

```
.iso.3.dod.1.mgmt.mib.1.sysDescr
```

All Agent Development Kit programs and utilities prepend the value `.iso.org.dod.internet.` to any relative object identifiers.

The Agent Development Kit automatically generates a sorted cross-reference of numeric OIDs to MIB object names whenever an agent executable is built. Refer to Chapter 4, “Developing an SNMP Agent,” for more information.

## Object Identifier Instance Indexes

To use the Agent Development Kit utilities, such as `snmpget`, to obtain values of objects, it is necessary to specify the instance of the object. The instance of an object is specified by appending an instance index to the object identifier. For example, the last 0 in:

```
.iso.3.dod.1.mgmt.mib.1.sysUpTime.0
```

is the instance index. An instance index of “0” (zero) specifies the first instance, “1” specifies the second instance, and so on. Since `sysUpTime` is a scalar object, it has only one instance. Therefore, an instance index of zero is always specified when retrieving the value of a scalar object. An instance index higher than 0 can only be used in the case of columnar objects, which can have multiple instances.

## SNMP Traps

In addition to retrieving data from the managed resource in response to requests from a management station, agents typically also have the ability to send unsolicited messages to managers when they detect some significant event. An unsolicited message of this sort is called a *trap notification*. The fields that comprise the SNMP trap packet occur in the order shown in Figure 2-5.

**Figure 2-5 SNMP Trap Protocol Data Unit (PDU)**

|          |            |               |                   |                    |           |                   |
|----------|------------|---------------|-------------------|--------------------|-----------|-------------------|
| PDU type | enterprise | agent address | generic trap type | specific trap type | timestamp | variable bindings |
|----------|------------|---------------|-------------------|--------------------|-----------|-------------------|

The fields have the following meaning:

- ◆ *PDU type* identifies the packet as a trap notification.
- ◆ *enterprise* is the vendor identification (OID) for the network management subsystem that generated the trap.
- ◆ *agent address* is the IP address of the node where the trap was generated.
- ◆ *generic trap type* is an integer in the range of 0 to 6. These values have the meanings indicated in Table 2-1.
- ◆ *specific trap type* is a number that further specifies the nature of the event that generated the trap in the case of traps of generic type 6 (enterpriseSpecific). The interpretation of this code is vendor-specific.
- ◆ *timestamp* is the length of time between the last re-initialization of the agent that issued the trap and the time at which the trap was issued.
- ◆ *variable bindings* provide additional information pertaining to the trap. This field consists of name/value pairs. The significance of this field is vendor-specific.

The interpretation of generic trap types is described in Table 2-1.

**Table 2-1 SNMP Generic Trap Types**

| <b>Generic Trap Number</b> | <b>Name of Trap Type</b> | <b>Interpretation</b>   |
|----------------------------|--------------------------|---|
| 0                          | coldStart                | The sending agent is re-initializing itself, typically due to a reboot.   |
| 1                          | warmStart                | The sending agent is re-initializing itself, typically due to a normal restart.   |
| 2                          | linkDown                 | One of the communication links on the agent node has failed. The first element in the variable bindings contains the name and value of the ifIndex instance for the downed interface. |
| 3                          | linkUp                   | One of the communication links on the agent node has come up. The first element in the variable bindings is the name and value of the ifIndex instance for the affected interface.    |
| 4                          | authenticationFailure    | The agent is reporting it has received a request with an invalid community specification or a community with insufficient permissions to complete the request.                        |
| 5                          | egpNeighborLoss:         | The agent is reporting that the peer relationship between an External Gateway Protocol (EGP) neighbor and an EGP peer no longer exists.   |
| 6                          | enterpriseSpecific       | The sending agent has detected an enterprise-specific event. The value of the specific trap type field indicates the nature of the event.   |

The Agent Development Kit provides an application programming interface (API) for generating traps from within your agent. The library of functions comprising this API are described in Chapter 3, “Tools and Functions.”

A sample program, `snmp_appl.c`, illustrating the use of the SNMP trap API, is provided in the `$RELEASE_ROOT/agentsrc/agentlib/` directory. A sample makefile, `sample.mk`, is also provided in the same directory to compile the sample program.

# Anatomy of SNMP Agents

SNMP agents or subagents built using the Agent Development Kit software are composed of the following components:

## Agent core

This layer contains the code to do SNMP and SMUX protocol handling and is independent of the specifics of the MIB supported by the agent. This layer contains the SNMP/SMUX protocol stack, the functions for traversing the OID tree, and the ASN.1 encoding and decoding functions.

## Agent hooks

This layer provides the mapping between the agent core layer and the access functions layer. Hooks map an SNMP or SMUX protocol request for a managed object to the appropriate access function.

## Access functions

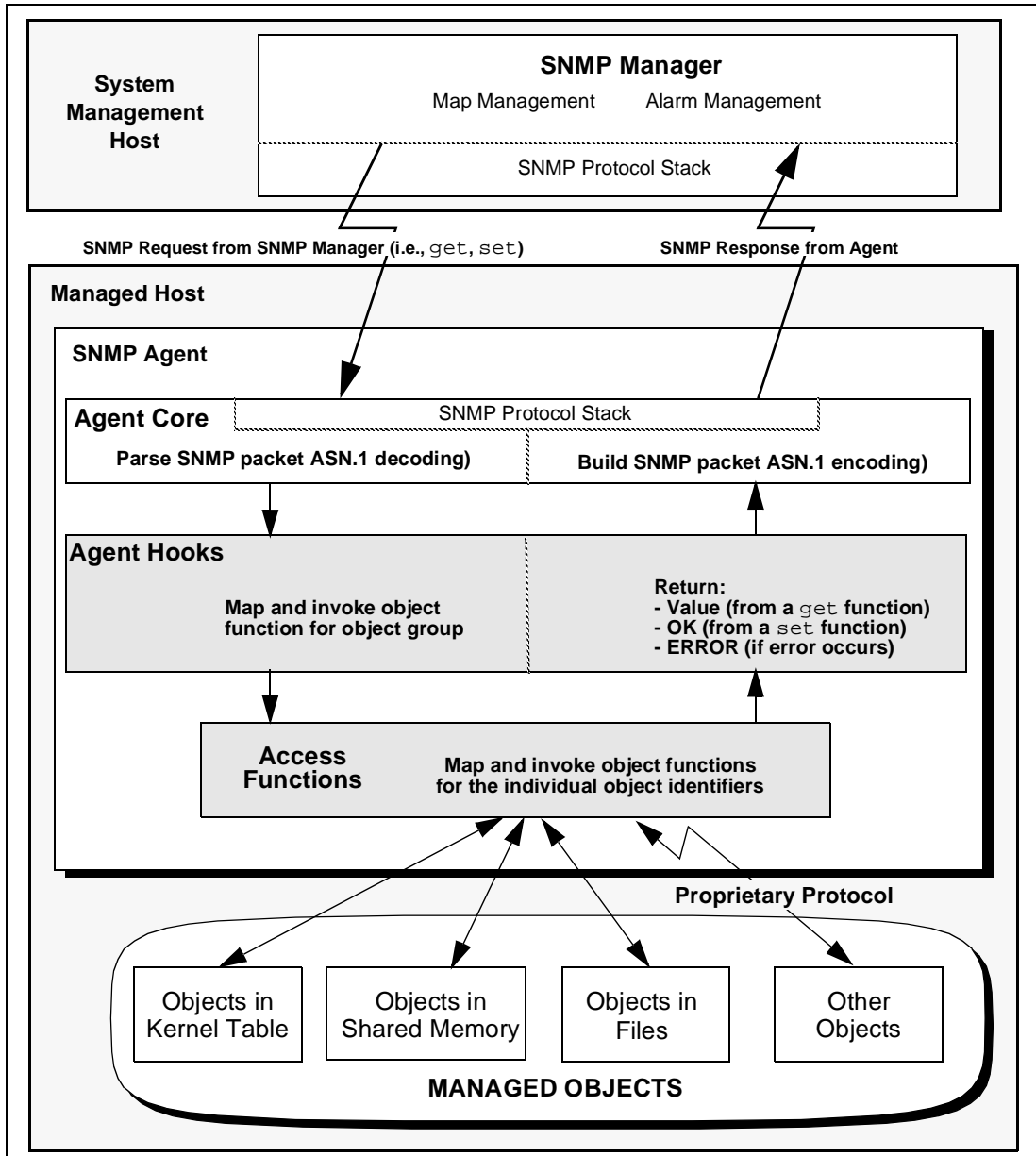
This layer is independent of the SMUX and SNMP protocols and is very dependent on the specific features of the supported MIB. These are functions that retrieve and change the values of attributes of the managed resource represented by MIB objects, thus providing the backend work to support SNMP GET and SET requests. The Agent Development Kit code generator creates skeletons of these functions which are then completed by the agent developer.

Because the Agent Development Kit builds the agent hooks and agent core layers for you, you need only be concerned with filling in the access function skeletons to retrieve or modify the values of the resource represented by the managed objects. The Agent Development Kit builds code for you that does the necessary SNMP and SMUX protocol handling and ASN.1 decoding and encoding, and mapping of SNMP requests to your custom access functions.

Figure 2-6 illustrates how these components fit together to process requests from, and send responses to, SNMP managers.



Figure 2-6 Data Flow to and from the Agent



## How the Agent Works

An agent built using the Agent Development Kit communicates with an SNMP manager, on the one hand, and the managed resource, on the other hand, in the following way:

1. The SNMP manager sends an SNMP packet to the agent. This packet contains an encoded request (such as a request to GET the value, or SET the value, of a specific managed object).
2. When the agent core receives the request, it parses the SNMP packet (ASN.1 decoding) to determine the type of request (i.e., GET or SET) and the MIB group, and invokes the appropriate agent hook.
3. The agent hook invokes the function associated with the SNMP MIB group to which the object belongs. The agent hook then invokes the access function corresponding to the object within the MIB group.
4. The access function does the actual work of retrieving the object's current value (for a GET request) or modifying the object's value (for a SET request). The method used to perform the GET or SET depends on where the managed object resides (i.e., UNIX kernel, shared memory, file, or in another process) and does not involve SNMP. If the object resides in another process, you can use shared memory or a proprietary protocol, such as Sun RPC/XDR or DCE RPC.

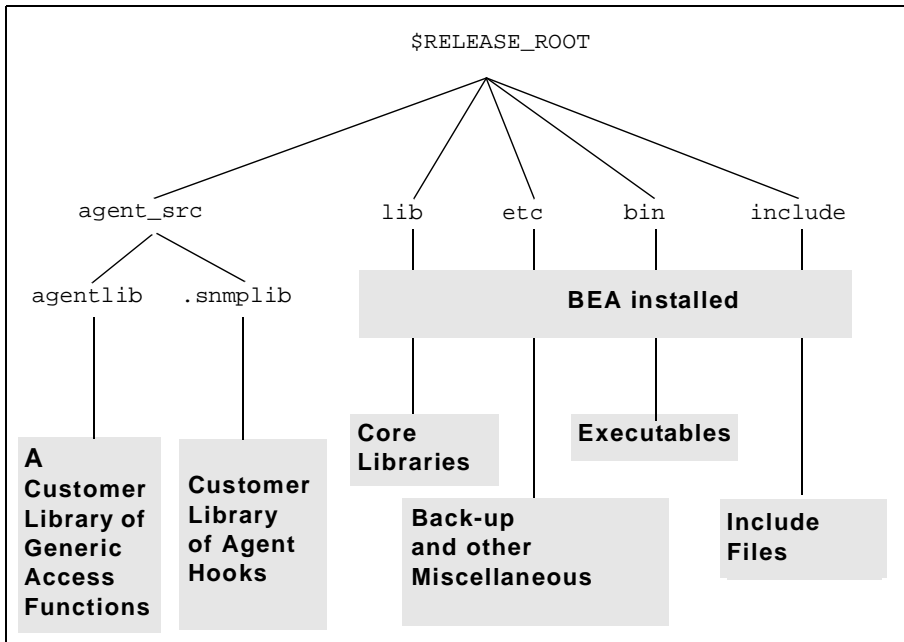
Depending on the value received from the access function, the agent hook layer returns one of the following responses to the agent core:

- ◆ A value from a GET function
  - ◆ An OK from a SET function
  - ◆ An ERROR, if an error occurred
5. The agent core receives the response from the agent hook and builds the SNMP packet (ASN.1 encoding). It then sends the response packet to the SNMP manager.

# Directory Structure

You can find the Agent Development Kit files in the `$RELEASE_ROOT` directory. Most customer development files will be found in the `agentlib` directory.

**Figure 2-7 Directory Structure**



**Note:** The items that appear in the gray boxes are descriptions only. They are not actual file names.

# Core Libraries

You need to include the core libraries and include files from the release when you build the SNMP agent or SMUX subagent. The code and declarations that accomplish this have already been included in the makefiles provided with this release. You will normally be working under the `$RELEASE_ROOT/agent_src/agentlib` directory.

# 3 Tools and Functions

This chapter describes the tools and application programming interfaces (APIs) included in the Agent Development Kit. It includes the following sections:

- ◆ “Introduction,” lists the tools and functions available with the Agent Development Kit and provides general information about the Agent Development Kit software.
- ◆ “Agent Tools,” describes the tools included with the Agent Development Kit software.
- ◆ “Agent Functions,” describes the library of functions available to the programmer for use in creating SNMP agents.

## Introduction

The Agent Development Kit provides a library of functions that comprise an application programming interface (API) and a set of utilities for building and testing agents or subagents. The Agent Development Kit tools and functions enable you to:

- ◆ Build an agent or subagent that supports a management information base (MIB).
- ◆ Query objects via your SNMP agent.
- ◆ Generate SNMP traps from your agent.

These tools and functions are listed in the following table.

| Agent Tools | Agent Functions        |
|-------------|------------------------|
| build_agent | csam_get_keyword       |
| imibgenall  | csam_set_keyword       |
| imibprint   | csam_trap_add_var_bind |
| instsrv     | csam_trap_create       |
| snmpget     | csam_trap_send         |
| snmpgetnext | bea_sendtrap           |
| snmptest    | bea_snmptrap           |
| snmptrap    |                        |
| snmptrapd   |                        |
| snmpwalk    |                        |

## MIB Variable Definition Files

When a MIB variable is used with an Agent Development Kit utility, the utility attempts to convert the variable to a numeric OID by searching first in a file named `mib.txt` in the current directory, and then in a file specified in the environment variable `BEA_SM_SNMP_MIBFILE`, and finally in the `/etc/mib.txt` file. These files should use ASN.1 notation and use the `OBJECT TYPE` macro defined in RFC 1155 (Structure of Management Information).

The `$RELEASE_ROOT/agent_src/agentlib/mib.txt` file describes the RFC 1213 (MIB-II) and the BEA private MIB objects.

## SNMP Request Format

Agent Development Kit utilities use SNMP requests to query SNMP agents for information about managed objects. Refer to RFC 1157 (SNMP) for more information about the format of SNMP requests. For information about locating RFCs on the Internet, refer to Appendix A, “SNMP Information,” in the *Agent Integrator Reference Manual*.

---

# Agent Tools

The Agent Development Kit software provides several tools to help you build and test an agent or subagent. The following list describes these tools.

|                          |   |
|--------------------------|---|
| <code>build_agent</code> | Builds SNMP and SMUX agent executables.   |
| <code>imibgenall</code>  | Generates or updates skeletal access function and hooks source code to support a private MIB group. |
| <code>imibprint</code>   | Prints a list of the MIB object identifiers and names.  |
| <code>instsrv</code>     | Installs an agent as a Windows NT service.  |
| <code>snmpget</code>     | Reports information about scalar managed objects.   |
| <code>snmpgetnext</code> | Returns the next entry in a table or the next consecutive managed object in a MIB.                  |
| <code>snmptest</code>    | Selectively performs <code>get</code> and <code>set</code> operations on any MIB object.            |
| <code>snmptrap</code>    | Sends an SNMP trap message to a host.   |
| <code>snmptrapd</code>   | Receives and logs SNMP trap messages sent on a local machine to the <code>snmp-trap</code> port.    |
| <code>snmpwalk</code>    | Traverses the OID tree using the SNMP <code>getnext</code> request to query managed objects.        |

## build\_agent

|             |   |
|-------------|---|
| Purpose     | Builds SNMP and SMUX agents.  |
| Synopsis    | <code>build_agent [ -q ] [ -n AgentName   -r MIBRootName ]</code>   |
| Arguments   | <p><code>-q</code></p> <p>The utility does not prompt for user input if this option is specified.</p> <p><code>-n AgentName</code></p> <p>The name of the agent executable that will be generated. Newly generated agents are placed in:</p> <p><code>\$RELEASE_ROOT/agent_src/agentlib</code></p> <p><code>-r MIBRootName</code></p> <p>This option specifies the MIB group name used when creating the access function module for stand-alone testing. The name of the built module is:</p> <p><code>csnmp_MIBRootName</code></p> <p>The <code>-r</code> and <code>-n</code> options are mutually exclusive.</p>  |
| Description | <p>If no options are specified, the utility enters agent creation mode with the default name of the agent set to <code>snmpd</code>.</p> <p>The first time an agent is built, the <code>build_agent</code> utility prompts the user for the MIB groups to be included, any ancillary files to be compiled, and any ancillary library files to be linked with the agent executable. This information is stored by the <code>build_agent</code> utility. If this agent is rebuilt, the current configuration is displayed and the utility prompts the user for any changes. A newly created agent is placed in the <code>\$RELEASE_ROOT/agent_src/agentlib</code> directory.</p> <p>The <code>build_agent</code> utility also builds a module of access functions for the specified MIB group. When created, the file is called <code>csnmp_MIBRootName</code> and is placed in the <code>\$RELEASE_ROOT/agent_src/agentlib</code> directory. This module can be used for testing the access functions in stand-alone mode.</p> |
| Examples    | <p>The command:</p> <pre>build_agent -n ex_snmpd -q</pre> <p>creates an agent with the name <code>ex_snmpd</code>.</p>  |

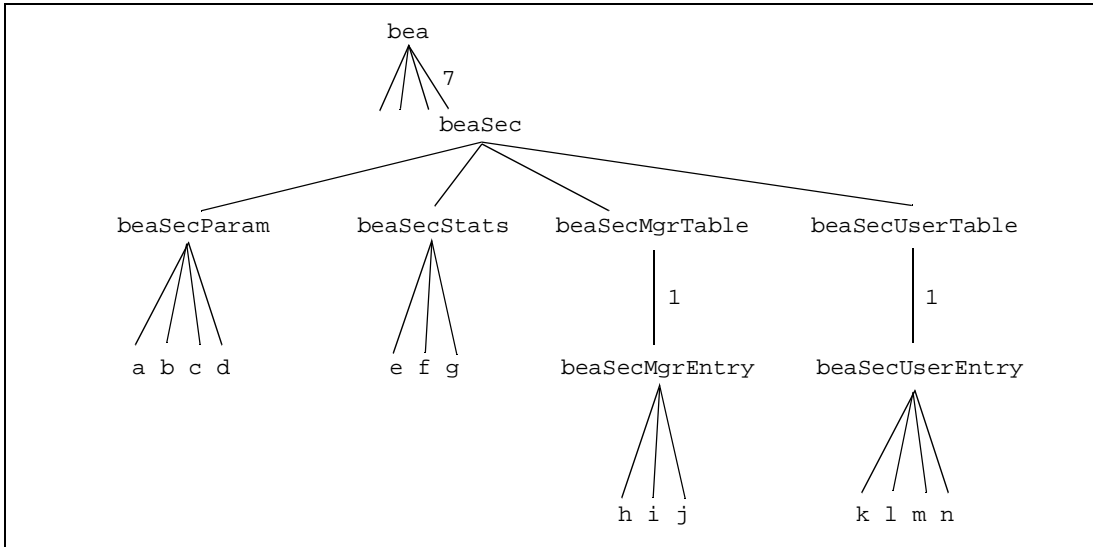


Because the `-q` option has been specified, `build_agent` will not go into interactive mode, asking which MIB groups need to be included in `ex_snmpd`. If the configuration is being created for the first time, `ex_snmpd` will be created with all the MIB groups currently developed. If this is not the first time the `ex_snmpd` agent has been built, the previously specified configuration is displayed but you will not be prompted for any changes.

|                       |   |
|-----------------------|---|
| Environment Variables | <code>RELEASE_ROOT</code><br>Contains the root directory of the Agent Development Kit installation. |
|-----------------------|---|

## imibgenall

|                       |   |
|-----------------------|---|
| Purpose               | Generates or updates all necessary agent code to support a new private MIB group  |
| Synopsis              | <code>imibgenall [ -u   -H <i>Headerfile</i> ] <i>MIBRootName</i></code>  |
| Arguments             | <p><code>-u</code><br/>Undo any code generation for the <i>MIBRootName</i> MIB group.</p> <p><code>-H <i>HeaderFile</i></code><br/>Specifies the file name containing customer-specific header information to be placed at the beginning of the generated files.</p> <p><i>MibRootName</i><br/>The root of the section of the MIB tree for which code will be generated.</p>  |
| Description           | <p>The <code>imibgenall</code> utility must be run from the <code>\$RELEASE_ROOT/agent_src/agentlib</code> directory.</p> <p>The <code>imibgenall</code> utility requires a <i>MIBRootName</i> argument. For scalar objects, indicate the name of the ancestor object (e.g., the name of the object above them in the tree). For tabular objects, indicate the name of the table object.</p> <p>By default, the <code>imibgenall</code> utility creates agent source code to support your private MIB. When the <code>-u</code> argument is specified, the utility removes any code previously generated for any portion of your MIB.</p> <p>If you are creating code to support your MIB, this utility creates several files to support the specified MIB in the SNMP agent: agent hook and access functions files, and their supporting header files. It also updates the makefiles that are needed to build an SNMP agent. You need only modify the generated access functions file:</p> <p><code>\$RELEASE_ROOT/agent_src/agentlib/<i>MIBRootName</i>.c</code></p> <p>where <i>MIBRootName</i> is the same as the argument passed to <code>imibgenall</code>. Once the code is created, your only concern is with filling in the access function skeletons in a single source file, and perhaps tuning of constants defined in its associated header file:</p> <p><code>\$RELEASE_ROOT/agent_src/agentlib/<i>MIBRootName</i>.h</code></p> |
| Environment Variables | <p><code>RELEASE_ROOT</code><br/>Contains the root directory of the Agent Development Kit installation.</p>   |
| Examples              | Assume a private MIB such as in the following diagram, where objects <code>a</code> through <code>g</code> are scalar, and objects <code>h</code> through <code>n</code> are columnar objects (i.e., scalar-valued objects that can have multiple instances).   |

**Figure 3-1 Sample Private MIB Tree**

The `imibgenall` utility must be run four times to create four agent hook files, their corresponding access function files, and other files. The four commands would be the following:

```
imibgenall beaSecParam
imibgenall beaSecStats
imibgenall beaSecMgrTable
imibgenall beaSecUserTable
```

The following creates code associated with the MIB group item `beaSecParam`:

```
imibgenall beaSecParam
```

Several files will be created. Usually, you will only need to modify the file `beaSecParam.c`, but you may be required to make some changes to the `beaSecParam.h` file. You can undo these operations by entering:

```
imibgenall -u beaEm
```

**Warning:** The `imibgenall -u` command will delete the `MIBRootName.c` and `MIBRootName.h` files. Before invoking `imibgenall` with this option, you may wish to make a backup copy of any changes in the `MIBRootName.c` and `MIBRootName.h` files that you want to preserve.

The `imibgenall -u` command will also delete any entries in the makefiles for the specified MIB group.

## imibprint

|                       |  |
|-----------------------|--|
| Purpose               | Prints a list of the MIB object identifiers and names from <code>mib.txt</code> .  |
| Synopsis              | <code>imibprint -o   -t   -sa   -so   -h</code>  |
| Arguments             | <div><div>-o</div><div>Prints the object identifiers, one per line.</div><div>-t</div><div>Prints the object identifiers in a tree structure, with each descendant OID indented one level from its ancestor OID.</div><div>-sa</div><div>Prints object identifiers, sorted by name.</div><div>-so</div><div>Prints object identifiers, sorted by numeric identifier.</div><div>-h</div><div>Prints the command synopsis.</div></div>                 |
| Description           | <code>mib.txt</code> is an ASCII text description of your MIB objects, generated from your ASN.1 MIB file by the code generator/MIB compiler.  |
| Environment Variables | <div>BEA_SM_SNMP_MIBFILE</div> <div>By default, <code>imibprint</code> first looks for <code>mib.txt</code> in <code>/etc</code>. If the file is not in that directory, <code>imibprint</code> uses the directory specified by this environment variable to locate <code>mib.txt</code>. If this environment variable is not set, and <code>mib.txt</code> is not in <code>/etc</code>, <code>imibprint</code> looks in the current directory.</div> |

## instsrv

**Purpose** Agents built using the Agent Development Kit can be installed as a Windows NT service using this utility. Supported on Windows NT machines only.

**Synopsis** `instsrv service-name [executable-file / remove]`

Enter *executable-file* to create a service. Enter *remove* to remove a service.

**Arguments** *service-name*

The name of the service. This should always be the same as the basename of the executable file.

*executable-file*

The complete path to the agent executable file.

## snmpget

|                       |  |
|-----------------------|--|
| Purpose               | Reports information about scalar managed objects.  |
| Synopsis              | <code>snmpget [-d] [-p <i>port</i>] <i>host</i> <i>community</i> <i>variable-name</i><br/>[<i>variable-name</i> ...]</code>  |
| Arguments             | <p><code>-d</code><br/>Causes the program to display a message for each packet.</p> <p><code>-p <i>port</i></code><br/>Specifies the port used to communicate with the SNMP agent (default: 161).</p> <p><code><i>host</i></code><br/>The internet address or host name of the node executing the SNMP agent to be queried.</p> <p><code><i>community</i></code><br/>The community name for the transaction.</p> <p><code><i>variable-name</i></code><br/>At least one unique object identifier (OID).</p>   |
| Description           | <p>The <code>snmpget</code> utility uses SNMP get requests to retrieve information about managed objects.</p> <p>You can enter one or more object identifiers as arguments on the command line. These names can be absolute, starting from the root of the tree, or relative to <code>.iso.org.dod.internet</code>. Read Chapter 2, “Agent Development Kit Overview,” for more information on specifying object identifiers.</p>   |
| Environment Variables | <p><code>BEA_SM_SNMP_MIBFILE</code><br/>Must be set to specify the path to <code>mib.txt</code>, which provides an ASCII text description of the contents of your private MIB.</p>   |
| Examples              | <p>The following command sends a query to the SNMP agent running on the host named <code>topaz</code>, using <code>public</code> as the community for authorization. The agent retrieves the value of the managed object <code>beaSysHasDisk</code> in the BEA private MIB. Note that in this example, a relative OID (<code>private.enterprises.bea.beaSystem</code>) is specified. <code>.iso.org.dod.internet.</code> is prepended to generate an absolute path.</p> <pre>snmpget topaz public private.enterprises.bea.beaSystem .beaSysHasDisk.0</pre> |

This command returns the following information about the object:

```
Name: private.enterprises.bea.beaSystem.beaSysHasDisk.0
INTEGER: yes(2)
```

The following command sends a query to the SNMP agent running on the host named `ruby`, using `public` as the community for authorization. The agent retrieves the value of the managed objects `sysDescr` and `sysUptime` in the MIB.

```
snmpget ruby public mgmt.mib.system.sysDescr.0
          mgmt.mib.system.sysUptime.0
```

This command returns the following information:

```
Name: mgmt.mib.system.sysDescr.0
OCTET STRING- (ascii): Kinetics FastPath2
```

```
Name: mgmt.mib.system.sysUptime.0
Timeticks: (2270351) 6:18:23
```



## snmpgetnext

**Purpose** Returns the next entry in a table or the next consecutive managed object in a MIB.

**Synopsis** `snmpgetnext [-d] [-p port] host community variable-name  
[variable-name ...]`

**Arguments** `-d`  
Causes the program to display a message for each packet.

`-p port`  
Specifies the port used to communicate with the SNMP agent (default: 161).

`host`  
The internet address or host name of the node executing the SNMP agent to be queried.

`community`  
The community name of the transaction.

`variable-name`  
At least one unique object identifier (OID).

**Description** You can enter one or more object identifiers as arguments on the command line. These names can be absolute, starting from the root of the tree, or relative to `.iso.org.dod.internet`. Read Chapter 2, “Agent Development Kit Overview,” for more information on specifying object identifiers.

**Environment Variables** `BEA_SM_SNMP_MIBFILE`  
Must be set to specify the path to `mib.txt`, which provides an ASCII text description of your private MIB.

**Examples** This example contacts the host named `blueberry` using the community name `public` and retrieves the value of the instance immediately following `mgmt.mib.interfaces.ifTable.ifEntry.ifOutOctets.0` from the MIB:

```
snmpgetnext blueberry public mgmt.mib.interfaces.ifTable.ifEntry
.ifOutOctets.0
```

**Note:** The instance index `.0` must be appended to the end of the OID to refer to the value of the object.

The output of the previous command might look like this:

```
Name: mgmt.mib.interfaces.ifTable.ifEntry.ifOutOctets.1
COUNTER: 85655250
```

You could then enter a command that retrieves information about the next variable:

```
snmpgetnext blueberry public mgmt.mib.interfaces.ifTable.ifEntry  
    .ifOutOctets.1
```

---

## snmpctest

**Purpose** Selectively performs get, getnext and set operations on any MIB object.

**Synopsis** `snmpctest [-d] [-p port] host community`

**Arguments:** `-d`

Causes the program to display a message for each packet.

`-p port`

Specifies the port used to communicate with the SNMP agent (default: 161).

`host`

The internet address or host name of the node executing the SNMP agent to be queried.

`community`

The community name of the transaction.

**Description** When this program is executed, it prompts you to enter an OID. The `snmpctest` utility returns information about request and reply packets as well as the name and type of the object. Read Chapter 2, “Agent Development Kit Overview,” for more information on specifying object identifiers.

By default, the program sends a GET request packet. This can be changed by entering a value from the following table at the prompt.

| Command | Request Type |
|---------|--------------|
| \$G     | GET          |
| \$N     | GETNEXT      |
| \$S     | SET          |

If you choose the SET request mode, the program prompts you for a variable type from the following list.

| Variable Type | Description  |
|---------------|--|
| a             | IP address.  |
| d             | Octet string as decimal bytes separated by white space (i.e., 105 118 105 101 119) |
| i             | Integer  |
| n             | Null value   |
| o             | Object identifier  |
| s             | Octet string in ASCII (i.e., bea)  |
| t             | Time ticks   |
| x             | Octet string as hexadecimal bytes separated by white space (i.e., 69 76 69 65 77)  |

After specifying the request type, the program prompts you to enter a value of the type you just specified. At this prompt, enter the integer (in decimal) or enter a string and press **Return**. To send the request packet, press **Return** again at the next prompt.

To quit the program, enter:

\$Q

Environment Variables      BEA\_SM\_SNMP\_MIBFILE  
Must be set to specify the path to `mib.txt`, which provides an ASCII text description of your private MIB.

Examples      Start the program by entering the command:

```
snmpctest topaz public
```

The program responds with:

```
Please enter the variable name:
```

Enter a variable name and press **Return**:

```
private.enterprises.bea.beaEm.beaEmMonitorTimer.0
```

The program requests another variable name:

Please enter the variable name:

You can either enter another variable name, or press **Return** to see the result. When you press **Return**, the program displays the result of the test:

```
Received GET RESPONSE from 192.84.232.47
requestid 0x775efba0 errstat 0x0 errindex 0x0
Name: private.enterprises.bea.beaEm.beaEmMonitorTimer.0
INTEGER: 5000
```

After displaying the result, you can enter another variable name, or \$Q to quit the program.

Please enter the variable name: \$Q

If you enter \$Q, a quit message displays:

```
Quitting, Good-bye
```

## snmptrap

|           |   |
|-----------|---|
| Purpose   | Sends an SNMP trap message to a host.   |
| Synopsis  | <code>snmptrap [-a <i>agent-addr</i>] [-d] [-p <i>port</i>] <i>host</i> <i>community</i> <i>trap-type</i> <i>specific-trap</i> <i>variable-binding-value</i></code>   |
| Arguments | <p><code>-a <i>agent-addr</i></code><br/>Specifies an originating address, if different from the host, where <code>snmptrap</code> is executed. This allows you to send a trap on behalf of another host.</p> <p><code>-d</code><br/>Causes the program to display a message for each packet.</p> <p><code>-p <i>port</i></code><br/>Specifies the port to which the SNMP trap should be sent on the target host (default is port 162).</p> <p><i>host</i><br/>The Internet address or name of the host to which the SNMP trap is to be sent.</p> <p><i>community</i><br/>The community name of the transaction.</p> <p><i>trap-type</i><br/>An integer that specifies the generic type (in the range 0 to 6) of the trap to be sent.</p> <p><i>specific-trap</i><br/>An integer that identifies the enterprise-specific trap that occurred when <i>trap-type</i> is set to generic trap type 6.</p> <p><i>variable-binding-value</i><br/>Information to be transported within the trap packet. The program uses this as the value in the variable binding list when it sends the trap.</p> |

Description This table defines the valid (generic) trap types.

| Name of Trap Type     | Generic Trap Number | Description  |
|-----------------------|---------------------|--|
| coldStart             | 0                   | The sending agent is re-initializing itself, typically due to a reboot.  |
| warmStart             | 1                   | The sending agent is re-initializing itself, typically due to a normal restart.  |
| linkDown              | 2                   | One of the communication links on the agent node has failed. The first element in the variable bindings contains the name and value of the ifIndex instance for the downed interface.    |
| linkUp                | 3                   | One of the communication links on the agent node has come up. The first element in the variable bindings contains the name and value of the ifIndex instance for the affected interface. |
| authenticationFailure | 4                   | The agent is reporting it has received a request with an invalid community specification or a community with insufficient permissions to complete the request.                           |
| egpNeighborLoss       | 5                   | The agent is reporting that the peer relationship between an External Gateway Protocol (EGP) neighbor and an EGP peer no longer exists.  |
| enterpriseSpecific    | 6                   | The sending agent is reporting that an enterprise-specific event has occurred. The value of the <i>specific-trap</i> field indicates the nature of the event.                            |

The trap generated by this tool has a fixed variable-binding list that contains only one object-value pair. The object is:

```
.iso.org.dod.internet.private.enterprises.bea.beaSystem.  
  beaTrapDescr.0
```

The value of this object may be specified in the *variable-binding-value* argument.

The enterprise field, which is part of the SNMP trap PDU header, is always:

```
.1.3.6.1.4.1.140.1.1
```

Which is equivalent to:

```
.iso.org.dod.internet.private.enterprises.bea.beaSystem.sysDescr
```

Read Chapter 2, “Agent Development Kit Overview,” for more information about the SNMP trap PDU.

**Examples** The following command sends a `coldStart` trap to the host named `topaz`, using `public` as the community for authorization. Note that a value for the *specific-trap* argument must be present, even though it is ignored when the value of the *trap-type* argument is not 6 (`enterpriseSpecific`).

```
snmptrap topaz public 0 1 "host xyz is booting"
```



## snmptrapd

|                       |  |
|-----------------------|--|
| Purpose               | Receives and logs SNMP trap messages sent to the snmp-trap port.   |
| Synopsis              | <code>snmptrapd [-d] [ -l <i>port</i> ] [-p]</code>  |
| Arguments             | <p><code>-d</code></p> <p>Causes the program to display a debug message for each packet.</p> <p><code>-l <i>port</i></code></p> <p>Specifies the port to use when listening for incoming trap packets (default is port 162).</p> <p><code>-p</code></p> <p>Causes the program to print trap information output to the standard output.</p>   |
| Environment Variables | <p><code>BEA_SM_SNMP_MIBFILE</code></p> <p>Must be used to specify the path to <code>mib.txt</code>, which provides an ASCII text description of your private MIB.</p>   |
| Description           | <p>This utility receives SNMP traps sent on the port specified by the <code>-l</code> argument. If no port is specified, it uses port number 162. This utility must be able to open the snmp-trap port, which usually requires <code>root</code> permissions.</p> <p>On UNIX platforms, if the <code>-p</code> argument is not specified, <code>snmptrapd</code> uses the UNIX <code>syslog</code> utility to log messages with a status of <code>WARNING</code>. If the <code>LOG_LOCAL0</code> facility is available, it is used instead of <code>syslog</code> or <code>snmptrapd</code>.</p> <p>On machines running Windows NT, if the <code>-p</code> argument is not specified, the NT Event Log is used to log <code>WARNING</code> messages.</p> |
| Examples              | <p>This command collects the incoming SNMP trap sent by another host, and displays it to standard output:</p> <pre>snmptrapd -p</pre> <p>When the host receives the trap, it displays the following information:</p> <pre>192.84.232.47: Cold Start Trap (0) Uptime: 0:00:00 Name: private.enterprises.bea. beaSystem.beaTrapDescr.0 OCTET STRING- (ascii): host xyz is booting</pre>  |

## snmpwalk

|                       |  |
|-----------------------|--|
| Purpose               | Traverses the OID tree using the SNMP <code>getnext</code> request to query managed objects.   |
| Synopsis              | <code>snmpwalk [-d] [-p port] host community [variable-name ...]</code>  |
| Arguments             | <p><code>-d</code><br/>Causes the program to display a message for each packet.</p> <p><code>-p port</code><br/>Specifies the port used to communicate with the SNMP agent (default: 161).</p> <p><code>host</code><br/>The host name or an Internet address, in “dot-dot” notation (i.e., separated with periods), where the SNMP request is to be sent.</p> <p><code>community</code><br/>The community name to use in the SNMP request.</p> <p><code>variable-name</code><br/>This is the unique object identifier, expressed symbolically, decimally, or as a combination of both. If you do not specify a variable name, <code>snmpwalk</code> searches the entire MIB.</p> |
| Description           | This utility traverses the OID tree from the object specified on the command line. You can enter one or more object identifiers as arguments on the command line. These names can be absolute, starting from the root of the tree, or relative to <code>.iso.org.dod.internet</code> . Read Chapter 2, “Agent Development Kit Overview,” for more information on specifying object identifiers. If no objects are specified, <code>snmpwalk</code> searches the entire MIB tree supported by the SNMP agent.   |
| Environment Variables | <p><code>BEA_SM_SNMP_MIBFILE</code><br/>Must be used to specify the path to <code>mib.txt</code>, which provides an ASCII text description of your private MIB objects.</p>  |
| Diagnostics           | <p>If the tree search causes the program to search beyond the end of the MIB, this message is displayed:</p> <p>End of MIB</p>   |
| Examples              | <p>This is an example of an <code>snmpwalk</code> command:</p> <pre>snmpwalk blueberry public private.enterprises.bea.beaSystem</pre>  |

This is some of the output generated from the command:

```
Name: private.enterprises.bea.beaSystem.beaSysSysname.0  
OCTET STRING- (ascii): SunOS
```

```
Name: private.enterprises.bea.beaSystem.beaSysNodename.0  
OCTET STRING- (ascii):  blueberry
```

# Agent Functions

There are three interfaces available that can be used to send an SNMP trap:

- ◆ `sendtrap`
- ◆ `snmptrap`
- ◆ Combined use of `csam_trap_create`, `csam_trap_add_var_bind`, and `csam_trap_send`

The simplest, but least flexible is `sendtrap`. This interface can only send enterprise-specific traps, but allows you to set the specific trap type and trap content. The trap host and community are taken from the `beamgr.conf` configuration file. The `sendtrap` function is actually a special case of the next interface, the `snmptrap` function.

The second interface, `snmptrap`, is more flexible, allowing you to specify an increased number of SNMP trap fields, but does not allow you to specify the variable binding list. A fixed variable (`private.enterprises.bea.beaSystem.beaTrapDescr.0`) is always sent as part of the trap. You can specify the value of this object as the trap description.

The third interface is the most powerful and flexible way of sending SNMP traps. Using the `csam_trap_create`, `csam_trap_add_var_bind`, and `csam_trap_send` functions, you can specify any part of an SNMP trap and include any number of variables in the variable binding list.

All of these interfaces are described in detail in this section.

| Function                            | Description   |
|-------------------------------------|---|
| <code>csam_get_keyword</code>       | Returns the value of a keyword                        |
| <code>csam_set_keyword</code>       | Sets the value of a keyword.                          |
| <code>csam_trap_add_var_bind</code> | Specifies the variable binding list for an SNMP trap. |
| <code>csam_trap_create</code>       | Creates the header portion of an SNMP trap.           |
| <code>csam_trap_send</code>         | Sends an SNMP trap to a host.                         |
| <code>bea_sendtrap</code>           | Sends SNMP enterprise-specific traps only.            |
| <code>bea_snmptrap</code>           | Sends an SNMP trap.                                   |

## csam\_get\_keyword

**Purpose** Returns the value of a keyword

**Synopsis**

```
#include <asn1.h>

csam_get_keyword(keyword, value)
char *keyword;
char *value;
```

**Arguments** *keyword* The string to match against the first token of each record (maximum length, 40 characters).

*value* A pointer to a buffer where a string of token(s) following the matched keyword is returned (maximum length, 200 characters).

**Return Values** The `csam_get_keyword` function returns one of the following values:

`SNMP_SUCCESS`

This function has executed successfully.

`SNMP_FAILURE`

This function has not executed successfully.

**Description** The `csam_get_keyword` function reads a record from the BEA Manager configuration file `beamgr.conf` (located in the `/etc` directory by default) based on the keyword passed to the function and returns the corresponding value. This function should be called only from the access functions or from the code which will be linked with the agent.

Each record in `/etc/beamgr.conf` is a newline-delimited keyword/value pair. A record can be continued on the next line by terminating the line with a backslash character. The backslash must be immediately followed by a newline character.

It is possible for multiple entries with the same keyword to exist in the `beamgr.conf` file. When the `csam_get_keyword` function is used with a specific keyword, it returns the value corresponding to the first matching entry in the file. Subsequent calls to `csam_get_keyword` with the *keyword* argument set to NULL retrieve the value corresponding to the next record matching the previously specified keyword. If there are no additional records that contain the previously specified keyword, the function fails, returning `SNMP_FAILURE`.

You need to make sure that the buffer pointed to by the *value* argument has sufficient space to store the returned value.

|             |  |
|-------------|--|
| Environment | BEA_SM_BEAMGR_CONF   |
| Variables   | May be used to specify the path to a BEA Manager configuration file (default: <code>/etc/beamgr.conf</code> ). |

## csam\_set\_keyword

**Purpose** Sets the value of a keyword.

**Synopsis**

```
#include <asn1.h>

csam_set_keyword(keyword, value)
char *keyword;
char *value;
```

**Arguments** *keyword* The string to match against the first token of each record (maximum length, 40 characters).

*value* A pointer to a buffer where a string of tokens following the matched keyword will be returned (maximum length, 200 characters).

**Return Values** The `csam_set_keyword` function returns one of the following values:

`SNMP_SUCCESS`

This function has executed successfully

`SNMP_FAILURE`

This function has not executed successfully.

**Description** The `csam_set_keyword` function sets the value of a keyword in the BEA Manager configuration file, `beamgr.conf` (located by default in `/etc`), and returns the corresponding value. This function should be called only from the access functions or from the code which will be linked with the agent.

Each record in `/etc/beamgr.conf` is a newline-delimited keyword/value pair. A record can be continued on the next line by terminating the line with a backslash character. The backslash must be immediately followed by a newline character.

**Note:** It is possible for multiple entries with the same keyword to exist in the `beamgr.conf` file. The `csam_set_keyword` function only sets the first instance of a specific keyword.

You need to make sure that the buffer pointed to by the *value* argument has sufficient space to store the returned value.

**Environment** `BEA_SM_BEAMGR_CONF`

**Variables** May be used to specify the path to a BEA Manager configuration file (default: `/etc/beamgr.conf`).

## csam\_trap\_create

**Purpose** Creates the header portion of an SNMP trap.

**Synopsis**

```
#include <stdio.h>
#include <snmp_user.h>
int csam_trap_create (community, EnterpriseOID, EnterpriseOIDLen,
    trapID, specificTrapID, agentAddr)
char *community;
oid *EnterpriseOID;
int EnterpriseOIDLen;
int trapID;
int specificTrapID;
char *agentAddr;
```

**Parameters** *community*

A pointer to the SNMP community name.

If the *community* parameter is NULL, the community defined in the TRAP\_HOST entry in the beamgr.conf file is used. If beamgr.conf cannot be located or the TRAP\_HOST entry is not defined in beamgr.conf, public is used as the default.

*EnterpriseOID*

The enterprise specific OID.

If the *EnterpriseOID* is a NULL pointer, then the OID defined in the SYSOBJID entry in the beamgr.conf file is used. If the beamgr.conf file cannot be located or SYSOBJID entry is not defined in the beamgr.conf file, then the default enterprise specific OID (.1.3.6.1.4.1.140.1.1) is used. This number is the BEA Systems enterprise specific OID.

*EnterpriseOIDLen*

The length of the *EnterpriseOID* in the sub-OID.

*trapID*

A pointer to the generic trap type, an integer in the range 0 to 6. The meaning of these trap types is described in Table 3-1.

*specificTrapID*

This should contain a meaningful number if *trapID* is set to 6.

*agentAddr*

The host name of the system purportedly sending this trap.



**Return Values** The `csam_trap_create` function returns one of the following values:

`CSAM_NOT_RIGHT_STATE`

This function has been used in an incorrect order.

`CSAM_INTERNAL_ERR`

There has been an internal error.

`CSAM_SUCCESS`

This function has executed successfully.

**Description** This function can be used to create the header of an SNMP trap within an agent program or other application. The `csam_trap_add_var_bind` function is used to specify the variable binding list of the trap and the `csam_trap_send` function is used to send the trap.

The value of the `trapID` parameter indicates one of the following (generic) trap types.

**Table 3-1 Generic Trap Ids**

| Name of Trap Type                  | Generic Trap Number | Description   |
|------------------------------------|---------------------|---|
| <code>coldStart</code>             | 0                   | The sending agent is re-initializing itself, typically due to a reboot.   |
| <code>warmStart</code>             | 1                   | The sending agent is re-initializing itself, typically due to a normal restart.   |
| <code>linkDown</code>              | 2                   | One of the communication links on the agent node has failed. The first element in the variable bindings contains the name and value of the <code>ifIndex</code> instance for the downed interface.    |
| <code>linkUp</code>                | 3                   | One of the communication links on the agent node has come up. The first element in the variable bindings contains the name and value of the <code>ifIndex</code> instance for the affected interface. |
| <code>authenticationFailure</code> | 4                   | The agent is reporting it has received a request with an invalid community specification or a community with insufficient permissions to complete the request.  |
| <code>egpNeighborLoss</code>       | 5                   | The agent is reporting that the peer relationship between an External Gateway Protocol (EGP) neighbor and an EGP peer no longer exists.   |
| <code>enterpriseSpecific</code>    | 6                   | The sending agent is reporting that an enterprise-specific event has occurred. The value of the <code>specific-trap</code> field indicates the nature of the event.                                   |

If any other number is given, it is treated as 6, and a value of 90000 is used in place of *specificTrapID*.

If the value of the *agentAddr* parameter is NULL, the name of the host where the program using this function is used. This argument can be set to some other value, allowing an agent program to send a trap on behalf of another host.

**Example** The following example creates the trap PDU header with a trap community name of public, generated on a host named bigguy:

```
if (( ret = csam_trap_create("public" /* Community */
                             ,bea /* Enterprise OID */
                             ,7 /* Length of the enterprise OID */
                             ,6 /* Generic trap ID */
                             ,123 /* Specific trap ID */
                             ,"bigguy" /*Host that generates trap */
                             )) != CSAM_SUCCESS)
    printf("Trap Create Failed with Error %d\n", ret);
```

A generic trap ID of 6 identifies this trap as an enterprise-specific trap. The length of the enterprise OID is the number of nodes in the path to the object from the root of the OID tree.

## csam\_trap\_add\_var\_bind

**Purpose** Specifies the variable binding list for an SNMP trap.

**Synopsis**

```
#include <stdio.h>
#include <snmp_user.h>

int csam_trap_add_var_bind(type, oidptr, oid_len, value, value_len)
int type;
oid *oidptr;
int oid_len;
union obj_val value;
int value_len;
```

**Parameters**

*type* The type of the object.

*oidptr* The OID of the object being sent.

*oid\_len* The number of nodes in the absolute path from root.

*value* The value of the object. It is of the *obj\_val* union type as defined in the file `snmp_user.h`.

*value\_len* The length of the *value* parameter.

**Return Values** The `csam_trap_add_var_bind` function returns one of the following values:

CSAM\_NOT\_RIGHT\_STATE  
This function has been used in an incorrect order.

CSAM\_WRONG\_TYPE  
The *type* argument is not valid.

CSAM\_INTERNAL\_ERR  
There has been an internal error.

CSAM\_SUCCESS  
This function has executed successfully.

**Description** After the `csam_trap_create` function has been called, this function can be called one or more times.

The `type` parameter must have one of the following values.

| Type      | value_len                                |
|-----------|--|
| OPAQUE    | Length in bytes                          |
| STRING    | Length in bytes                          |
| IPADDRESS | Length in bytes                          |
| OBJID     | Number of nodes in the absolute OID path |
| INTEGER   | Ignored                                  |
| COUNTER   | Ignored                                  |
| GAUGE     | Ignored                                  |
| TIMETICKS | Ignored                                  |

**Examples** The following example adds an integer value to the variable bindings of a trap PDU:

```
value.int_val = 812;
if ( (ret = csam_trap_add_var_bind(INTEGER /*Object type      */
                                ,oid1 /* OID of the object   */
                                ,10  /* Length of the OID    */
                                ,value /* Object value       */
                                ,sizeof(int) /* Size of the object value */
                                ) != CSAM_SUCCESS)
    printf("Add to Trap Var Bind Failed with Error %d\n", ret);
```

The length of the value is not significant in the case of integer values. The length of the OID is the number of nodes from root to the specified object in the path through the OID tree. In the next example, a string value is added to the trap variable bindings:

```
trap_str_val = "Device is not responding";
value.str_val = trap_str_val;
if ( (ret = csam_trap_add_var_bind(STRING, oid2, 10, value,
                                strlen(trap_str_val) /*Length of value in bytes */
                                ) != CSAM_SUCCESS)
    printf("Add to Trap Var Bind Failed with Error %d\n", ret);
```

In the following example, an object identifier is added to the trap variable bindings:

```
value.oid_val = trap_oid_val;
if (( ret = csam_trap_add_var_bind(OBJID, oid3, 10, value,
    9 /* Length of the object */
    ) ) != CSAM_SUCCESS)
    printf("Add to Trap Var Bind Failed with Error %d\n", ret);
```

The length of the OID is nine in this case.

## csam\_trap\_send

**Purpose** Sends an SNMP trap to a host.

**Synopsis**

```
#include <stdio.h>
#include <snmp_user.h>
int csam_trap_send( hostname, port )
char *hostname;
int port;
```

**Parameters**

*hostname*  
The host to which the trap is sent.

*port*  
The UDP port to which this trap is sent.

**Return Values** The `csam_trap_send` function returns one of the following values:

`CSAM_NOT_RIGHT_STATE`  
This function has been used in an incorrect order.

`CSAM_GETHOSTNAME_FAILED`  
Invocation of `gethostname` to determine current host name failed.

`CSAM_ASN_ENCODING_FAILED`  
ASN.1 encoding failed.

`CSAM_SOCKET_OP_FAILED`  
Socket operation failed.

`CSAM_BIND_OP_FAILED`  
Bind operation failed.

`CSAM_HOST_NOTOK`  
The host name to which the trap is being sent is not valid.

`CSAM_SENDTO_FAILED`  
Send failed.

`CSAM_INTERNAL_ERR`  
There has been an internal error.

`CSAM_SUCCESS`  
Function executed successfully.

**Description** This function must be called after the `csam_trap_create` and `csam_trap_add_var_bind` functions.

If the `hostname` argument is `NULL`, the host name specified by the `TRAP_HOST` entry in the `beamgr.conf` is used. If `beamgr.conf` cannot be located, or the `TRAP_HOST` entry is not defined, the local host name is used. If there are multiple `TRAP_HOST` entries, the trap will be sent to each host specified in all `TRAP_HOST` entries.

If the `port` argument is 0, the port number specified by the `TRAP_HOST` entry in the `beamgr.conf` file is used. If `beamgr.conf` cannot be located, or the `TRAP_HOST` entry is not defined, the standard SNMP trap port 162/udp is used.

**Example** In the following example a trap is sent to the host `comet` on port 162:

```
if (( ret = csam_trap_send("comet" /* Name of destination host */
                          ,162    /* Destination port number    */
                          )) != CSAM_SUCCESS )
    printf("Trap Send Failed with Error %d\n", ret);
```

The destination host name is the name of a machine running an SNMP manager or an SNMP trap daemon, or the BEA `snmptrapd` utility.

## bea\_sendtrap

|               |  |
|---------------|--|
| Purpose       | Sends an SNMP trap.  |
| Synopsis      | <pre>#include &lt;snmp_user.h&gt; int bea_sendtrap (specificType, trapDesc) int specificType; char *trapDesc;</pre>  |
| Parameters    | <p><i>specificType</i><br/>Because the generic trap type is enterprise-specific, this parameter identifies more accurately the type of trap.</p> <p><i>*trapDescr</i><br/>A pointer to some data describing the trap. This parameter is always transmitted in the MIB object:<br/><br/> <pre>private.enterprises.bea.beaSystem.beaTrapDescr.0</pre> </p>   |
| Return Values | <p>The <code>bea_sendtrap</code> function returns one of the following values:</p> <p><code>SNMPERR_NOERR</code><br/>No error occurred.</p> <p><code>SNMPERR_CANNOT_OPEN</code><br/>Cannot open SNMP port.</p> <p><code>SNMPERR_CANNOT_SEND</code><br/>Cannot send SNMP packet.</p> <p><code>SNMPERR_CANNOT_OPEN_CONF_FILE</code><br/>Cannot open the SNMP configuration file to read the host name (<code>TRAP_HOST</code> entry) that receives the trap. The default file is <code>/etc/beamgr.conf</code> or is defined by the environment variable <code>BEA_SM_BEAMGR_CONF</code>.</p> <p><code>SNMPERR_INVALID_PARAMS</code><br/>Invalid parameters were passed.</p> |
| Description   | <p>The <code>bea_sendtrap</code> function takes an integer and a character pointer as input. All other SNMP trap fields are predefined. It is a specific case of <code>snmptrap</code>, in which the gateway and the community are extracted from the <code>beamgr.conf</code> file and the trap type is 6 (i.e., enterprise-specific).</p> <p>The <i>trapDescr</i> pointer is always transmitted in the MIB object:<br/><br/> <pre>private.enterprises.bea.beaSystem.beaTrapDescr.0</pre> <p>The trap is sent on the standard SNMP trap port 162/udp.</p> </p>  |



**Example** In the following example, an enterprise-specific trap is sent to the host name specified by the TRAP\_HOST keyword in the `beamgr.conf` configuration file:

```
if (( ret = bea_sendtrap(2244 /* Specific Trap ID */
                        , "ZYX queue is clogged" /* Trap description */
                        )) != SNMPERR_NOERR )
    printf("Trap Send Failed with Error %d\n", ret);
```

## bea\_snmptrap

**Purpose** Sends an SNMP trap.

**Synopsis**

```
#include <snmp_user.h>
int bea_snmptrap ( gateway, community, trapType, specificType,
                  trapDesc, agentAddr )
char *gateway, *community, *trapType, *specificType, *trapDescr,
      *agentAddr;
```

**Parameters**

*gateway*  
A pointer to the address of the host where the trap is to be sent. It can be a host name or an IP address in the “dot-dot” notation.

*community*  
A pointer to the SNMP community name.

*trapType*  
A pointer to the generic trap type.

*specificType*  
A pointer to a specific trap type. It is applicable only for enterprise-specific trap type, (i.e., generic trap type 6). It is present even if the generic trap type is not enterprise-specific.

*trapDescr*  
A pointer to some data describing the trap.

*agentAddr*  
The name or the IP address of the host that is assumed to send the SNMP trap. If it is passed as NULL, the local host name is used.

All parameters are essential. This function has no defaults.

**Return Values** The `bea_snmptrap` function returns one of the following values:

`SNMPERR_NOERR`  
No error occurred.

`SNMPERR_CANNOT_OPEN`  
Cannot open SNMP port.

`SNMPERR_CANNOT_SEND`  
Cannot send SNMP packet.

SNMPERR\_CANNOT\_OPEN\_CONF\_FILE

Cannot open the SNMP configuration file.

BEA\_SNMPERR\_INVALID\_PARAMS

Invalid parameters were passed.

**Description**     *trapDescr* is always sent as the value of the same predefined object. It is always transmitted in the MIB object:

```
private.enterprises.bea.beaSystem.beaTrapDescr.0
```

The trap is sent on the standard SNMP trap port 162/udp.

**Examples**     The sample program `snmp_appl.c` that makes use of this function (and a sample makefile for use with `snmp_appl.c`) can be found in the `$RELEASE_ROOT/agent_src/agentlib` directory.



# 4 Developing an SNMP Agent

This chapter explains how to develop an SNMP agent or SMUX subagent to manage a system or network resource. It includes the following sections:

- ◆ “Overview of the Development Process,” describes the necessary steps in the development of an SNMP agent using the Agent Development Kit.
- ◆ “Adding Your Custom Code to Generated Files,” describes the individual steps in writing access functions for your MIB.
- ◆ “Testing Your Custom Code,” describes how to test your custom access functions code before linking with the SNMP agent.
- ◆ “Building the Agent Executable,” describes the individual steps in building the agent executable.
- ◆ “Testing the Agent,” describes the steps in testing the agent executable.
- ◆ “Using Multiple SNMP Agents,” discusses how you can run your SNMP agent if the host has an existing SNMP agent.

# Overview of the Development Process

The Agent Development Kit lets you build an SNMP agent to support manageable features of a system or network resource defined in your own MIB (private or not private). The steps involved in developing an SNMP agent or SMUX subagent, using the Agent Development Kit, follow:

1. **Research and conceptualize, or form a model of, the system or network resource to be managed. You will want to select, or pinpoint, those features of the resource that are pertinent to system and network management goals.**

2. **Define the MIB for the objects to be managed.**

This step involves assigning variable names to each distinct manageable item, and determining and assigning the ASN.1 data types. The MIB must be written in ASN.1 notation. Refer to RFC 1212 (Concise MIB Definitions) for information about the syntax of defining MIB groups. Before defining a MIB, you may wish to obtain your own enterprise object identifier (OID). Refer to Appendix A, “SNMP Information,” in the *Agent Integrator Reference Manual* for instructions on how to obtain an enterprise OID.

One approach to writing a MIB is to start with an existing MIB, selecting features similar to those you want, and modifying the entries appropriately. Accordingly, you might want to look at some examples, such as standard MIBs or the BEA MIBs in the `bea.asn1` file. You must define your MIB in the file `my.asn1`. You can edit this file with any text editor, such as `vi`. The path name of the file is:

```
$RELEASE_ROOT/agent_src/agentlib/my.asn1
```

The Agent Development Kit code generator (`imibgenall`) checks for correct ASN.1 syntax and generates appropriate messages if it encounters syntax errors.

3. **The application or component being managed requires instrumentation that provides information in a form that can be accessed by the agent or subagent.**

The information must be provided in a way that matches the definitions in the MIB. For example, a queueing system might contain a counter that collects data on the number of current entries in a queue. An application programming interface (API) function call might provide this information in the form of an integer value.

#### 4. Define your environment (if you have not yet done so)

The following is an example of how to define the working environment on a UNIX platform, using C shell commands:

```
setenv RELEASE_ROOT install_dir
set path = ($RELEASE_ROOT/bin $path)
```

The following is an example of how to define the working environment on a Windows NT system:

```
set RELEASE_ROOT = install_dir
set path = %RELEASE_ROOT%\bin;%PATH%
```

All other necessary environment settings are set automatically.

#### 5. Run the MIB compiler/code generator (*imibgenall*) on a selected table or MIB group in your MIB file.

The MIB compiler generates function skeletons to help you build your SNMP agent. The compiler also creates other source files, which you do not need to modify, which provide “hooks” to the access functions. The compiler also updates the makefiles that are needed to build the agent. To do this you use the compiler command:

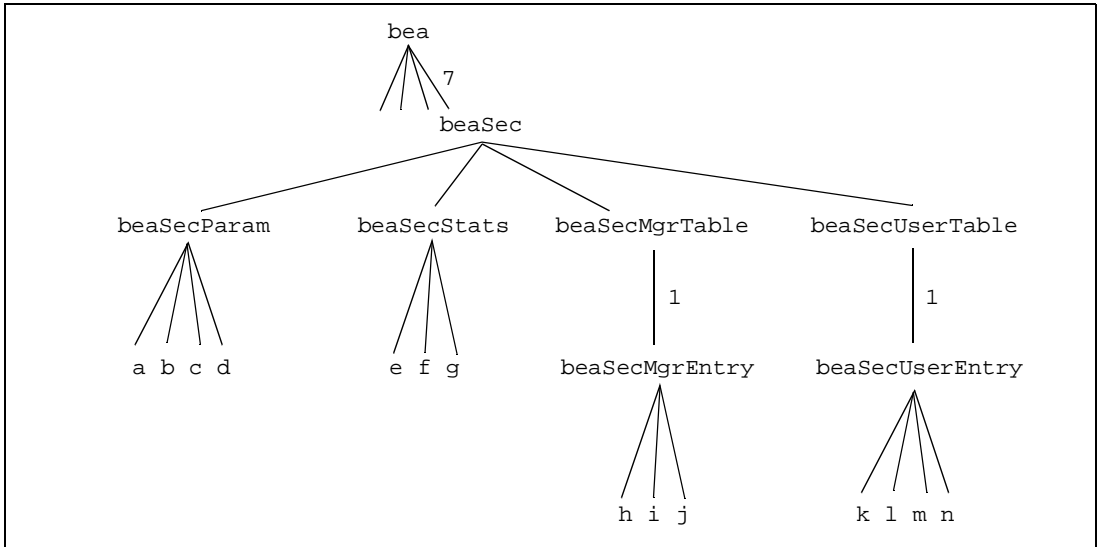
```
imibgenall MIBRootName
```

*MIBRootName* designates the target MIB component. (Refer to Chapter 3, “Tools and Functions,” for the complete syntax of this command.)

The target MIB component (*MIBRootName*) must not be a group of groups but either a table or, for scalar objects, their ancestor group (i.e., the group that contains the scalar objects). If you have multiple groups of scalar objects or both scalar objects and tables, you will need to run the compiler multiple times. You run the compiler once for each group that contains scalar objects and once for each table. The MIB compiler expects to find the MIB definitions in the file *my.asn1*.

Figure 4-1 shows an example of a private MIB. In this example, the objects contained in the groups *beaSecParam* and *beaSecStats* (objects *a* through *g*) are scalar objects whereas *beaSecMgrTable* and *beaSecUserTable* are tabular objects in the group *beaSec*. Objects *h* through *n* are columnar objects (i.e., objects that can have multiple scalar-valued instances).

Figure 4-1 Sample Private MIB Tree



The `imibgenall` utility must be run four times to create four agent hook files, their corresponding access function files, and other files. This could be accomplished by entering the following four commands:

```

imibgenall beaSecParam
imibgenall beaSecStats
imibgenall beaSecMgrTable
imibgenall beaSecUserTable

```

#### 6. Flesh out the generated function skeletons by adding code to access the data in the component or application being managed.

To do this you edit the source file `MIBRootName.c` generated by the MIB compiler. To flesh out the generated function skeletons, you use the instrumentation in the managed resource (referred to in step 3) to access the manageable features of the resource, to respond to SNMP GET and SET commands. Be sure that you create the file skeletons in the correct Agent Development Kit directory: `$RELEASE_ROOT/agent_src/agentlib`.

The specific work required to carry out this step is described in “Adding Your Custom Code to Generated Files.” The function skeletons and related components in the generated source files are described in Chapter 5, “Generated Access Function Templates.”



## 7. Do stand-alone testing of the access functions.

Once you have written the access functions (step 5), you can test them in stand-alone fashion, before linking with the rest of the agent code. To do this, compile the source file with the following command:

```
cd $RELEASE_ROOT/agent_src
build_agent -r MIBRootName
```

This builds an executable named `csnmp_MIBRootName`. This executable invokes the access functions and prints the values that are retrieved. To run this executable, enter the following command:

```
cd $RELEASE_ROOT/agent_src/agentlib
./csnmp_MIBRootName
```

## 8. Build the SNMP agent executable.

Use the following command to build the agent:

```
build_agent -n YourSNMPAgent
```

The `build_agent` command is interactive and prompts you for the following information:

- ◆ The MIB groups (corresponding to the files created in steps 4 through 5) to be included in the agent
- ◆ The name of any other source code files

**Note:** Any C files that you list here must be located in the following directory:

```
install_directory/agent_src/agentlib
```

- ◆ The name of any program-specific precompiled libraries

**Note:** You must use the absolute path to any libraries specified here.

- ◆ Any program-specific compilation flags

## 9. Test the SNMP agent using the SNMP test utilities.

The `snmpwalk` and `snmpptest` utilities can be used to test the agent, as described in “Testing the Agent.”

## 10. Integrate your private MIB with an SNMP management platform and test the agent with the management system.

You need to follow the vendor instructions of your SNMP manager to integrate a new MIB into the manager.

Figure 4-2 illustrates the three basic phases in the development process.

**Figure 4-2 Writing, Building and Testing an Agent**

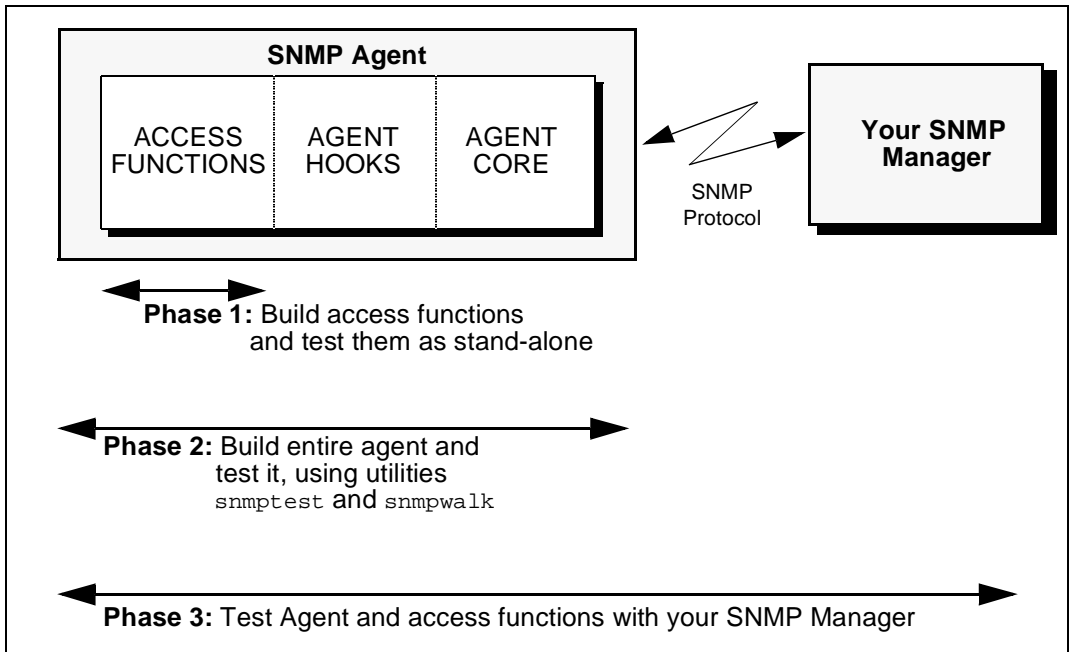
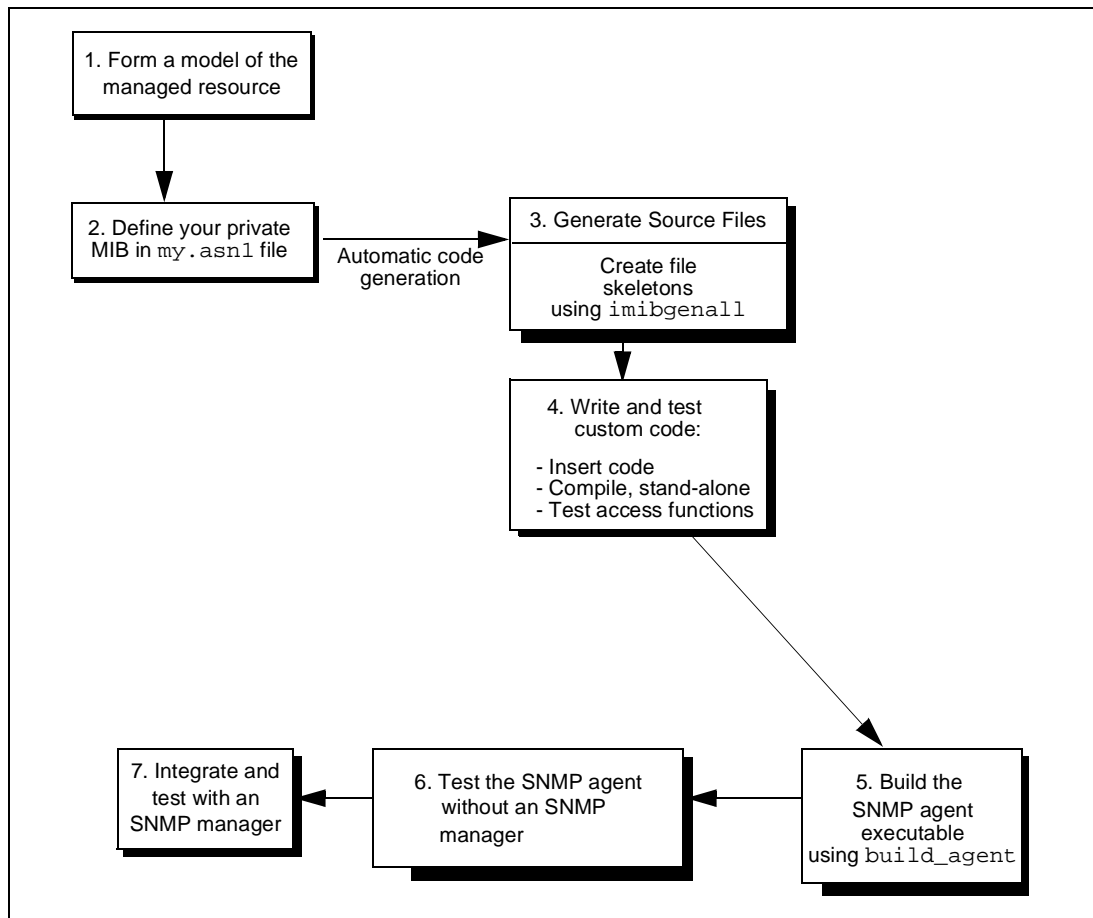


Figure 4-3 illustrates the detailed steps of the agent development process.

**Figure 4-3 Agent Creation Flowchart**



## Adding Your Custom Code to Generated Files

You create an access functions skeleton file and header file for each MIB group or table by running the MIB compiler/code generator (`imibgenall`). You need to run the compiler once for each group or table, by entering the command:

```
imibgenall MIBRootName
```

where *MIBRootName* is the name of a table or a group that contains scalar objects. The compiler expects to find your MIB definitions in the file *my.asn1*.

Be sure to generate the source files in the correct Agent Kit directory:

```
$RELEASE_ROOT/agent_src/agentlib.
```

The generated files that need to be modified are:

*MIBRootName.c* — generated access functions file

*MIBRootName.h* — generated header file

The access function skeletons and other components of these generated files are described in Chapter 5, “Generated Access Function Templates.”

Once you have generated these source files for a MIB group, you need to edit the files to carry out the following tasks:

### 1. Modify the constant definitions in the header file, if necessary.

The header file generated for each MIB group or table defines a constant that determines the maximum length for read-only and read/write string objects in the agent cache — *MAX\_ReadOnlyStringObjectName*. By default these values are set to 256. You may want to modify these values.

One of the constant values that gets defined for tabular MIB groups is *INIT\_MIBRootName\_ENTRIES*. By default, the value of this constant is 10. This constant represents the number of the entries in the cache for a table, space for which is allocated at initialization time. Another constant, *DELTA\_MIBRootName\_ENTRIES*, represents the number of new row entries for which space will be allocated if the currently allocated space for the cache table runs short.

You should set the *INIT\_MIBRootName\_ENTRIES* constant to the typical value of the number of entries in the table. For example, if you are writing a MIB for a process table, a good value of *INIT\_MIBRootName\_ENTRIES* can be 200 and if you are writing a MIB for disks, a good value of *INIT\_MIBRootName\_ENTRIES* would be 10. If you choose a big number for this constant, you will end up wasting memory. If you choose a small number, the SNMP agent may end up doing multiple allocations, which may fragment memory.

### 2. Use resource instrumentation in the refresh function to update the agent cache.

When the agent receives an SNMP GET or GETNEXT request from a management station, it calls the generated `get_ObjectName` functions to send back the current value of the object. The generated access functions file contains one such `get_ObjectName` function for each object in the MIB group or table. These `get_ObjectName` functions do not access the managed resource but retrieve the current value from the agent cache. Normally, you do not need to modify the `get_ObjectName` functions.

The agent tracks the age of the cache. Before calling the `get_ObjectName` functions, the agent checks to determine whether the cache is older than the refresh rate. If it is older, the agent calls the function `refresh_MIBRootName` before calling the `get_ObjectName` functions for that MIB group. This refresh function updates the agent cache by directly accessing the managed resource — for example, using an API or system calls — to obtain current values of the attributes of the managed resource. Accordingly, you must edit the `refresh_MIBRootName` function to add the code that directly accesses the managed resource and updates the object values in the agent cache.

For example, if your MIB group has a scalar object `myQEntryCount` under the `myQ` MIB group, there will be a line in the generated code such as the following:

```
v_myQ.val_myQEntryCount = dummy_value;
```

The object values for a given MIB group are cached in the record `v_MIBRootName` — in this example, `v_myQ`. This record is an instance of a structure of type `s_MIBRootName`, defined in the header file. You will need to replace `dummy_value` with the code that provides the actual value accessed from the managed resource — in this example, this might involve a call to the queue system's API.

For tabular objects, there can be zero or more rows (entries) in the table. Thus, you need to use some sort of loop control to fill out the values corresponding to each row in the table. This loop control requires some index that indicates the number of rows in the table. The dummy loop control code generated in the access functions file for tables looks something like this:

```
int row_num;
for (i=0; i<2; i++)
{ /* This dummy code assumes 2 rows in the table */
/* The following statement increments the size of the table, if required */
if (!create_MIBRootName_entry(&row_num)) return SNMP_FAILURE;
```

A function `create_MIBRootName_entry()` is generated for each tabular object. This function creates a row in a table. The function returns an index of the newly created row. You should use this index to fill out the values of the objects in this

row in the cache table. You must call `create_MIBRootName_entry()` each time in the control loop to create a row. This function should not be called from anywhere else in the agent program. Otherwise, the results are undefined. Before `refresh_MIBRootName()` is called by the SNMP agent, earlier rows are discarded. You can fill in the rows in the cache in any order. The SNMP agent automatically arranges them in lexicographic order in replying to the management station, as required by the SNMP standard.

### 3. Modify the refresh rate, if necessary.

The agent maintains an internal cache of the values of the managed objects for each of its MIB groups. The agent tracks the time since the last refresh of the cache. When the agent receives an SNMP GET or GETNEXT request from a management station, it takes the value from the cache if the cache is not older than the refresh rate, otherwise it invokes the refresh function to update the cache. The refresh function uses the instrumentation in the managed resource to update the agent cache that maintains current values for the managed objects accessed by the agent. The refresh rate is specified by the constant `MIBRootName_refresh_rate`. By default this value is set to 10 seconds. This is set by the following line in the generated access functions file:

```
int MIBRootName_refresh_rate = 10 /* seconds */;
```

You may want to increase this value if the object values in the managed resource do not change very quickly. If the value of `MIBRootName_refresh_rate` is set to zero, the refresh function is called every time an object is queried by a management station. The refresh function is not invoked if no requests are received by the agent for the MIB group objects.

**4. Flesh out the initialization function for the MIB group.**

An initialization function skeleton `init_MIBGroupName` is generated for each MIB group or table. This function is called once by the agent the first time any object in the MIB group is polled. You should fill in this function with any required initialization code. This function returns `SNMP_SUCCESS` if successful, and `SNMP_FAILURE` otherwise. If this function returns a failure, the agent or subagent always returns “NO SUCH NAME” for any object in this MIB group. It is recommended that your code print a reason for failure when failures occur.

**5. Edit the test functions to check values in set requests to ensure that the value is valid.**

If an SNMP agent receives an SNMP SET request containing more than one variable, either all objects are set or none are set. This behavior is known as *atomic set* and is one of the requirements of the SNMP standard. Atomic sets are supported using a two-phase commit. In order to implement this, for each read-write object a `test` and a `set` function is generated as part of the access functions file. On receiving a SET request from the management station, the SNMP agent first invokes the test function corresponding to each object involved in the SET request. If all `test` functions return success, only then does the agent invoke the `set` functions corresponding to these objects.

For each scalar or columnar object that is defined as read-write in the MIB, a `test_ObjectName` function skeleton is generated in the access functions file. You must flesh out each of these functions with code that checks to determine whether the value in the SET request is valid for the target attribute in the managed resource. The test function should return success only if the requested set value is valid for the target.

**6. Use appropriate access methods (e.g., APIs or system calls) in set functions to modify attributes of the managed resource.**

To support SNMP SET requests, a `set_ObjectName` function is generated in the access functions file for each MIB object that is defined as read-write. You need to add the necessary code to directly access the managed resource, such as API function calls. See “Row Deletion” and “Row Addition” in “Additional Programming Guidelines” for supporting SET requests for addition or deletion of rows in tables.

## Additional Programming Guidelines

The following points may be useful for writing the access functions code. In the following sections, *MIBRootName* refers to the root name of the MIB tree for which the C code is generated using `imibgenall`.

### Community

If the agent is working as an SNMP agent, access functions can access the SNMP community used by the management station (if needed). The community is made available as an `extern` character array `sid`. The integer `sidlen` contains the length in bytes.

### Defining Keywords in `beamgr.conf`

The SNMP agent uses a configuration file, `beamgr.conf`. Each record in `beamgr.conf` is separated by a newline character. Each record represents a keyword-value pair. The SNMP agent uses this file for maintaining certain configuration parameters and MIB object values. As the implementor of a MIB, you can use the same file for defining your own keywords.

To facilitate the use of this file, two APIs, `csam_get_keyword()` and `csam_set_keyword()`, are provided. `csam_get_word()` fetches the value corresponding to a given keyword and `csam_set_keyword()` sets the value corresponding to a keyword in `beamgr.conf`.

For more details about these two APIs, please refer to Chapter 3, “Tools and Functions.”

### Sample Traps Program

The Agent Development Kit includes APIs for generating traps from your SNMP agent. The trap APIs are described in Chapter 3, “Tools and Functions.” The directory `$RELEASE_ROOT/agentsrc/agentlib/` contains a sample program, `snmp_appl.c`, illustrating the use of the SNMP trap APIs. A sample makefile, `sample.mk`, is also provided in the same directory to compile the sample program.



## Row Deletion

In order to support row deletion capability, you must define an integer-valued read-write columnar object, which represents the status of the row. This object minimally has two valid values representing `valid` and `invalid` status. When the management station sets the value of this object to `invalid`, the effect should be of invalidating the corresponding row from the table.

To help the implementation of row deletion effects, an API `delete_MIBRootName_entry()` is provided. This function deletes a row from cache maintained by SNMP agent for the *MIBRootName* MIB group.

It should be noted that `delete_MIBRootName_entry()` only deletes the row from cache and has no effect on the source of the MIB information (e.g., kernel memory). In order to remove a row from the actual place of MIB information, you should take specific action inside the SET function corresponding to this MIB object, which is generated as part of the access functions file.

The function `delete_MIBRootName_entry()` expects the row index to be deleted, as an argument. This row index is the one that is passed to the set routine. `delete_MIBRootName_entry()` returns `SNMP_SUCCESS`, if the row deletion is successful, else it returns `SNMP_FAILURE`.

**Note:** This function must be called only from a set routine generated as part of the access functions file. If it is called from any other place, results are undefined.

## Row Addition

In order to create a new row, the management station issues a SNMP set request containing one or more variables that refers to an object instance not currently available in the SNMP agent.

In order to support row creation, two functions, `test_MIBRootName_create()` and `set_MIBRootName_row()`, are created as part of the access functions file.

These two functions are called when a SET request is received for a non-existent row. First, all values corresponding to different columnar objects are stored in the cache, then `test_MIBRootName_create()` is called to check if the values are acceptable. If it returns success, `set_MIBRootName_row()` is called. `set_MIBRootName_row()` should always return success for atomic sets to work correctly.

Both of these functions are passed the index of the newly created row in the cache.

If `test_MIBRootName_create()` returns failure, row creation is assumed to have failed and the newly created row is removed from the cache.

It should be noted that a `set` request from the management station may not contain all columnar objects. It is the responsibility of the `test_MIBRootName_create()` function to figure out if this set of columnar objects is sufficient to create a row. The objects that are received in the set request are indicated by an `SNMP_VALID` value of their corresponding status. In the process, programmers may assign some suitable default values to other columnar objects in `test_MIBRootName_create()` and `set_MIBRootName_row_create()`.

## Testing Your Custom Code

To test the access function file `MIBRootName.c`, compile it as stand-alone code to bypass the SNMP requests. To do this, enter the following command:

```
cd $RELEASE_ROOT/agent_src
build_agent -r MIBRootName
```

This generates the file:

```
$RELEASE_ROOT/agent_src/agentlib/csnmp_MIBRootName
```

To run the executable, enter:

```
cd $RELEASE_ROOT/agent_src/agentlib/
./csnmp_MIBRootName
```

This executes the get functions and prints the retrieved values. This ensures that the access functions work as a stand-alone utility before linking it with the rest of the SNMP agent code.

# Building the Agent Executable

Follow these steps to build the agent source code:

1. Change to the directory containing the agent source code:

```
cd $RELEASE_ROOT/agent_src
```

2. Execute the `build_agent` command:

```
build_agent -n SNMPAgentName
```

The `build_agent` command prompts you for additional information:

- ◆ The MIB groups to be included in the newly created agent
- ◆ The name of any source code files other than *MIBRootName.c*
- ◆ The name of any program-specific precompiled libraries. This would consist of a list of absolute paths to the libraries, with entries separated by a blank.
- ◆ Any program-specific compilation flags

The `build_agent` command creates the agent executable in the file *SNMPAgentName* in the following directory:

```
$RELEASE_ROOT/agent_src/agentlib
```

3. This step applies to UNIX platforms only: if you want to run the agent executable as a stand-alone SNMP agent (`-s` option) after building the agent executable, do the following:

- ◆ Set the owner of the executable to `root`.
- ◆ Set the `s-uid` bit of the executable.

For example:

```
cd $RELEASE_ROOT/agent_src/agentlib
chown root SNMPAgentName
chmod u+s SNMPAgentName
ls -l SNMPAgentName
-rwsrwxr-x 1 root 499712 Jun 20 10:58 SNMPAgentName*
```

4. Add the following services to your `/etc/services` files or to the services file on your yp host, if you are running yellow pages.

|           |         |
|-----------|---------|
| snmp      | 161/udp |
| snmp-trap | 162/udp |

5. Run the agent:

```
./SNMPAgentName -s
```

The above command starts *SNMPAgentName* as an SNMP agent. If you want to run *SNMPAgentName* as a SMUX subagent, do not enter the `-s` command-line option. But in that case, a SMUX master agent or Agent Integrator should already be running.

The `beamgr.conf` file can be placed in the `/etc` directory, or you can set the `BEA_SM_BEAMGR_CONF` environment variable to point to this file so the agent can send a `coldstart` trap when it starts up. The default host is the local host.

For more details about command-line options of newly created agents, please refer to Chapter 5, “Starting the Subagents,” in the *Agent Integrator Reference Manual*.

## Object Identifier Lists

The `sorted_oid_list` and `ascii_oid_list` files are automatically generated whenever an SNMP agent is built using the `build_agent` utility. These files contain a sorted cross-reference mapping managed object MIB names to OIDs.

## Sample sorted\_oid\_list

Listing 4-1 is an example of the file `sorted_oid_list`, containing the `beaEx` example MIB objects.

### Listing 4-1 sorted\_oid\_list

---

```
..
beaTrapHost: 1.3.6.1.4.1.140.1.8
beaTrapCommunity: 1.3.6.1.4.1.140.1.9
beaEx: 1.3.6.1.4.1.140.100
beaExIntRO: 1.3.6.1.4.1.140.100.1
beaExIntRW: 1.3.6.1.4.1.140.100.2
beaExStrRO: 1.3.6.1.4.1.140.100.3
beaExStrRW: 1.3.6.1.4.1.140.100.4
beaUnix: 1.3.6.1.4.1.140.2
beaPsTable: 1.3.6.1.4.1.140.2.1
..
```

---

## Sample ascii\_oid\_list

Listing 4-2 is an example of the file `ascii_oid_list`, containing the `beaEx` example objects. In this example, the file contents look the same because object names and object identifiers are alphabetically and numerically in the same order.

### Listing 4-2 ascii\_oid\_list

---

```
..
beaEmProcsEnvVar: 1.3.6.1.4.1.140.4.3
beaEmShmAllocated: 1.3.6.1.4.1.140.4.4
beaEx: 1.3.6.1.4.1.140.100
beaExIntRO: 1.3.6.1.4.1.140.100.1
beaExIntRW: 1.3.6.1.4.1.140.100.2
beaExStrRO: 1.3.6.1.4.1.140.100.3
beaExStrRW: 1.3.6.1.4.1.140.100.4
..
```

---

# Testing the Agent

To test the SNMP agent executable set the environment variable `BEA_SM_SNMP_MIBFILE` to:

```
$RELEASE_ROOT/agent_src/agentlib/mib.txt
```

This ensures that the SNMP utilities can properly access the objects you added to the MIB file as the `mib.txt` file contains your new managed object definitions.

Test the agent by using either the `snmpwalk` or `snmpstat` utilities. To use the `snmpwalk` utility, enter (all on one line):

```
snmpwalk hostname public private.enterprises.bea.beaEx
```

where *hostname* is the name or Internet address (in dot-dot notation) of the node where the agent is running.

The following data is retrieved from the `snmpwalk` command:

```
Name: private.enterprises.bea.beaEx.beaExIntrO.0
INTEGER: 1
Name: private.enterprises.bea.beaEx.beaExIntrRW.0
INTEGER: 2
Name: private.enterprises.bea.beaEx.beaExStrRO.0
OCTET STRING- (ascii): get_beaExStrRO
Name: private.enterprises.bea.beaEx.beaExStrRW.0
OCTET STRING- (ascii): get_beaExStrRW
End of MIB.
```

**Note:** `snmpwalk` and `snmpstat` print the full English name of objects because they have access to the `mib.txt` file.

Refer to the Chapter 3, “Tools and Functions,” for complete instructions.

## Installing the New Agent

To install the new SNMP agent, copy the SNMP agent executable to the host where you want to use it. Set its s-uid bit and ownership to root.

Install the `beamgr.conf` and `bea_snmpd.conf` files and the following network services:

|           |         |
|-----------|---------|
| snmp      | 161/udp |
| snmp-trap | 162/udp |

The default location for the agent executable is `/usr/sbin` on SVR4 systems. The default location for `beamgr.conf` and `bea_snmpd.conf` is `/etc` on UNIX platforms. On Windows NT systems, the configuration files are located in `C:\etc`.

## Troubleshooting

There are several error messages you may encounter during the installation process:

bind: Address already in use

This message informs you that an SNMP agent is already running on that host. To correct this problem, terminate the conflicting agent.

bind: Permission denied

This message informs you that the user-id and the group-id of the agent are incorrect. To correct this problem, change the effective user-id of the agent to root.

bind: can't find snmp service

This message informs you that the SNMP services are not configured correctly. To correct this problem, make sure the following services are available:

|           |         |
|-----------|---------|
| snmp      | 161/udp |
| snmp-trap | 162/udp |

# MIB Modification

If you realize that you made a mistake in defining a new MIB after you have created the C code using the `imibgenall` utility, you can undo it with the following command:

```
imibgenall -u MIBRootName
```

For example:

```
imibgenall -u beaEx
```

will undo any `beaEx` MIB related sources/modifications (previously created using `imibgenall beaEx`).

You might want to do this for one of the following reasons:

- ◆ You want to add or delete some objects in the MIB group.
- ◆ You want to change the type of certain MIB objects in this group.
- ◆ You want to change the INDEX clause of the MIB group.
- ◆ You want to change the access level of certain MIB object (i.e., changing from read-only to read-write or vice-versa).

Perhaps you have written code in the access functions file already which you do not want to lose. In such a situation, follow these steps:

1. Save a copy of the access function file. For example, on UNIX platforms, enter the following commands:

```
cp MIBRootName.c MIBRootName.c.sav  
cp MIBRootName.h MIBRootName.h.sav
```

The following would be the Windows NT commands:

```
copy MIBRootName.c MIBRootName.c.sav  
copy MIBRootName.h MIBRootName.h.sav
```

2. Undo the earlier created sources corresponding to this MIB. To do so, type:

```
imibgenall -u MIBRootName
```

3. Make any desired modifications in the `MIBRootName` MIB group definitions in the `my.asn1` file.



4. Create the C code for this MIB group again. To do so, execute this command:

```
imibgenall MIBRootName
```

Most of your changes made to *MIBRootName.c.sav* should be usable as is in *MIBRootName.c*.

5. Delete *MIBRootName.c.sav*.

## Using Multiple SNMP Agents

If a host already has an SNMP agent (for example, an SNMP agent supplied by the hardware manufacturer), there are two strategies for using an agent generated by the Agent Development Kit:

- ◆ An agent generated using the Agent Development Kit can run on a port other than the standard SNMP port (161). If an SNMP manager can monitor two different SNMP agents on the same node using two different ports, you can use a port other than 161 for the SNMP manager to communicate with your agent. You can set the port for your SNMP agent with this command:

```
SNMPAgentName -s -p port_number
```

where *port\_number* is the alternate port. Configure the SNMP manager to communicate with this agent on *port\_number* according to the manufacturer's instructions.

- ◆ If your third-party SNMP manager cannot communicate with an SNMP agent on a port other than port 161, you can use the BEA Agent Integrator to listen on port 161, and handle manager communications with SNMP agents (and subagents). Refer to the NON\_SMUX\_PEER entry in Chapter 6, "Configuration Files," in the *Agent Integrator Reference Manual*.



# 5 Generated Access Function Templates

This chapter describes function templates and related elements created in the access functions file and header file that are generated by the Agent Development Kit code generator (`imibgenall`). The following topics are discussed:

- ◆ “Function Templates,” describes the access function skeletons that need to be fleshed out by the programmer to provide access to the managed resource.
- ◆ “Constants and Variables,” describes the constants and variables that are available for use or modification by the programmer in the access functions file.
- ◆ “Generated Functions,” describes the generated functions in the access functions file that normally do not need to be modified by the programmer.

Function *templates* are skeletons that require the programmer to fill in implementation-specific code. The access functions source contains that portion of the agent that accesses the attributes of the managed resource, to modify those attributes in response to a SET request from a manager, or to retrieve current values in response to a GET request from a manager. This code is specific to your implementation because it depends upon the particular instrumentation available in the managed resource, such as application-specific APIs or UNIX system calls.

To generate the source code file containing the access functions for your agent, you target the MIB compiler/code generator (`imibgenall`) at a MIB group or table with the following command:

```
imibgenall MIBRootName
```

where *MIBRootName* is the target MIB group or table.

When you run the MIB compiler/code generator with this command, the code generator reads the `my.asn1` file and generates access function skeletons corresponding to the target MIB objects. The code is created in a file named `MIBRootName.c`. The code generator also creates a declaration for a C structure that is used by the agent to cache the current values of the managed objects. This structure, `v_MIBRootName`, is declared in the header file, `MIBRootName.h`.

Normally the `v_MIBRootName` cache structure and the `get_ObjectName` functions do not need to be modified by the programmer.

## Function Templates

This section describes the function templates that are created by the code generator.

### **init\_MIBRootName**

This function is called once by the agent the first time any object in the `MIBRootName` group or table is polled. You need to fill in this function skeleton with your initialization code. It is recommended that you print a reason for failure in case `SNMP_FAILURE` is returned.

Syntax     `int init_MIBRootName( )`

Return Value     This function returns `SNMP_SUCCESS` if successful and `SNMP_FAILURE` if not successful. If this function returns a failure, the agent or subagent always returns “NO SUCH NAME” for any object in this MIB group or table.

### **refresh\_MIBRootName**

Syntax     `int refresh_MIBRootName( )`

Return Value     This function has no return value for MIB groups with scalar objects. For tabular objects, the function returns either `SNMP_SUCCESS` or `SNMP_FAILURE`. If `SNMP_FAILURE` is returned, no SNMP operation can be performed on objects in that table for the next refresh period.

One such refresh function is generated for the MIB group or table that is the target passed to the code generator when generating an access functions skeleton file.

This refresh function is responsible for updating the agent cache by directly accessing the managed resource — for example, using an API or system calls — to obtain current values of the attributes of the managed resource. Accordingly, you must edit the `refresh_MIBRootName` function to add the code that directly accesses the managed resource and updates the object values in the agent cache.

For example, if your MIB group has a scalar object `myQEntryCount` under the `myQ` MIB group, there will be a line in the generated code such as the following:

```
v_myQ.val_myQEntryCount = dummy_value;
```

The object values for a given MIB group are cached in the record `v_MIBRootName` — in this example, `v_myQ`. This record is an instance of a structure of type `s_MIBRootName`, defined in the header file. You will need to replace `dummy_value` with the code that provides the actual value accessed from the managed resource — in this example, this might involve a call to the queue system’s API.

For each field `val_ObjectName` in the cache structure, there is also a status field `sts_ObjectName`. For the previous example, there would be a line in the generated code that looks something like this:

```
v_myQ.sts_myQEntryCount = dummy_value;
```

This field should contain the status of the corresponding `val_ObjectName` field. If the value of `ObjectName` is not available, your code should set `v_MIBRootName.sts_ObjectName` to `SNMP_INVALID`. For example:

```
v_myQ.sts_myQEntryCount = SNMP_INVALID;
```

Also, for each field `val_ObjectName` in the structure that is of type string, there will be a corresponding field `len_ObjectName`, which should contain the length of the string in bytes.

The structures that constitute the agent’s cache of current object values are used by the `get_ObjectName` functions to retrieve object values in response to a GET request from a management station. The agent tracks the age of the cache. Before calling the `get_ObjectName` functions to respond to manager GET requests, the agent checks to determine whether the cache is older than the refresh rate. If it is older, the agent calls `refresh_MIBRootName` to update the cache before calling the `get_ObjectName` functions for that MIB group. For more information about the refresh rate, see the following section, “Constants and Variables.”

### test\_ObjectName

For each scalar or columnar object that is defined as read-write in the MIB, a `test_ObjectName` function skeleton is generated in the access functions file. You must flesh out each of these functions with code that checks to determine whether the value in the set request is valid for the target attribute in the managed resource.

The syntax of the test function depends upon the type of *ObjectName*.

**Syntax** For scalar objects that are OIDs, strings, or IP addresses:

```
int test_ObjectName(value, length)
u_char *value;
int length;
```

For scalar objects other than strings, OIDs or IP addresses:

```
int test_ObjectName(value)
int value;
```

For columnar objects that are strings, OIDs, or IP addresses:

```
int test_ObjectName(row_index, value, length)
int row_index;
u_char *val_ptr;
int length;
```

For columnar objects other than OIDs, strings, or IP addresses:

```
int test_ObjectName(row_index,value)
int row_index;
int value;
```

**Return Value** The test function should return `SNMP_SUCCESS` if and only if the requested set value is valid for the target. Otherwise, the function returns `SNMP_FAILURE`.

If an SNMP agent receives an SNMP SET request containing more than one variable, the agent either sets all the objects in the SET request or else none are set. This behavior is known as *atomic set* and is one of the requirements of the SNMP standard. Atomic sets are supported using two-phase commit. In order to implement this, for each read-write object a test and a set function is generated as part of the access functions file.

On receiving a SET request from the management station, the SNMP agent first invokes the test function corresponding to each object involved in the SET request. If all test functions return success, only then does the agent invoke the set functions corresponding to these objects. Otherwise, no set function is called.

For string objects or IP addresses, `length` is the length in bytes. For OIDs, `length` is the number of nodes in the absolute path from the root of the OID tree.

## set\_ObjectName

To support SNMP SET requests, a `set_ObjectName` function skeleton is generated in the access functions file for each scalar or columnar MIB object that is defined as read-write. You need to add the necessary code to directly access the managed resource, such as API function calls.

The syntax of this function depends upon the type of *ObjectName*.

**Syntax** For scalar objects other than OIDs, strings, or IP addresses:

```
int set_ObjectName(value)
int value;
```

For scalar objects that are strings, OIDs, or IP addresses:

```
int set_ObjectName(value, length)
u_char *value;
int length;
```

For columnar objects that are strings, IP addresses, or OIDs:

```
int set_ObjectName(row_index, value, length)
int row_index;
u_char *value;
int length;
```

For columnar objects other than strings, IP addresses, or OIDs:

```
int set_ObjectName(row_index, value)
int row_index;
int value;
```

**Return Value** This function should always return `SNMP_SUCCESS`.

The agent executes a `set_ObjectName` function if and only if a call to the corresponding `test_ObjectName` function returns `SNMP_SUCCESS`. The programmer must fill out the `set_ObjectName` function to directly access the managed resource to modify the specified attribute.

If an SNMP agent receives an SNMP set request containing more than one variable, the agent either sets all the objects in the SET request or else none are set. This “atomic set” behavior is one of the requirements of the SNMP standard. Atomic sets are

supported using two-phase commit. In order to implement this, for each read-write object a test and a set function is generated as part of the access functions file. On receiving a set request from the management station, the SNMP agent first invokes the test function corresponding to each object involved in the set request. If all test functions return success, only then does the agent invoke the set functions corresponding to these objects. This function must always return either `SNMP_SUCCESS` or `SNMP_FAILURE` for an atomic set to work properly.

For string objects or IP addresses, `length` is the length in bytes. For OIDs, `length` is the number of nodes in the absolute path from the root of the OID tree.

### test\_TableName\_row\_create

**Syntax**     `int test_TableName_row_create(row_index)`  
                  `int row_index;`

**Return Value**     Returns `SNMP_SUCCESS` if enough object values have been provided to set an entire row in the table and the values are appropriate for the target objects. Returns `SNMP_FAILURE` otherwise.

The agent cache maintains a record `v_TableName` for each row in a table. In this record, the field `val_ObjectName` stores the value of the columnar object while `sts_ObjectName` stores the status of the object. If the value of an object cannot be provided, the status should be assigned `SNMP_INVALID`. If `sts_ObjectName` is set to `SNMP_INVALID`, the corresponding value is ignored.

When a row creation request is received, the values are first loaded into the `v_TableName` record and then the agent calls the `test_TableName_row_create` to determine if the values are acceptable for the target objects. If this function returns `SNMP_SUCCESS`, `set_TableName_row_create` is called. Both of these functions are passed an index to the newly created row structure in the cache. If `test_TableName_row_create` returns `SNMP_FAILURE`, the newly created row structure is removed from the agent cache.

A SET request from a master agent to a subagent may fail to contain values for all of the columnar objects in the row. It is the responsibility of the `test_TableName_row_create` function to ensure that a sufficient number of columnar object values have been provided to create a row. The validity of the object values is indicated to the `set_TableName_create_row` function by the `sts_ObjectName` field corresponding to the object in the `v_TableName` structure being assigned a value of `SNMP_VALID`.



## set\_TableName\_row\_create

**Syntax**     `int set_TableName_row_create(row_index)  
              int row_index;`

**Return Value**     This function should always return `SNMP_SUCCESS`.

The agent cache maintains a record `v_TableName` for each row in a table. In this record, the field `val_ObjectName` stores the value of the columnar object while `sts_ObjectName` stores the status of the object. If the value of an object cannot be provided, the status should be assigned `SNMP_INVALID`. If `sts_ObjectName` is set to `SNMP_INVALID`, the corresponding value is ignored.

When a SET request for a non-existent row is received, the values are first loaded into the `v_TableName` record and then the agent calls the `test_TableName_row_create` to determine if the values are acceptable for the target objects. If this function returns `SNMP_SUCCESS`, `set_TableName_row_create` is called. Both of these functions are passed an index to the newly created row structure in the cache. If `test_TableName_row_create` returns `SNMP_FAILURE`, the newly created row structure is removed from the agent cache.

The validity of the object values is indicated to the `set_TableName_create_row` function by the `sts_ObjectName` field corresponding to the object in the `v_TableName` structure being assigned a value of `SNMP_VALID`.

# Constants and Variables

This section describes user-modifiable constant values and the generated agent cache structure variable, `v_MIBRootName`.

## DELTA\_TableName\_ENTRIES

This constant is defined in the header file, `MIBRootName.h`, when the code generator is targeted at a table. Each row in a table is stored in memory in a structure `v_TableName`, which is of type `s_TableName`. This type is defined in the `MIBRootName.h` header file. Objects within a table are stored in this structure. The number of rows in the table are dynamically adjusted. Whenever it is necessary to add more rows, the value of the constant `DELTA_TableName_ENTRIES` determines the amount of space that is added incrementally to provide space for additional rows.

## INIT\_TableName\_ENTRIES

This constant is defined in the header file, `MIBRootName.h`, when the code generator is targeted at a table. Each row in a table is stored in memory in a structure `v_TableName`, which is of type `s_TableName`. This type is defined in the `MIBRootName.h` header file. Objects within a table are stored in this structure. The number of rows in the table are dynamically adjusted. However, when the table is created it is initially allocated a number of rows equal to the constant `INIT_TableName_ENTRIES`.

## MAX\_StringObjectName

For objects of type string in the MIB group or table, a maximum string length constant is defined in the `MIBRootName.h` header file. For example:

```
#define MAX_beaExStrRO 256 /* length of beaExStrRO in bytes */
```

You can modify the value of this constant as appropriate.

## MIBRootName\_refresh\_rate

This variable specifies in seconds the smallest interval at which the refresh function, `refresh_MIBRootName`, is called within the agent. The value of this variable is set in the access functions source file, `MIBRootName.c`.

When a manager sends a GET request to the agent, `get_ObjectName` functions are called to retrieve the current value from the agent cache. The `get_ObjectName` functions do not directly access the managed resource. Direct access of the managed resource is the work of the `refresh_MIBRootName` function, which updates the agent cache. The agent tracks the age of the cache. Before calling the `get_ObjectName` functions, the agent checks to determine whether the cache is older than the refresh rate. If it is older, the agent calls the function `refresh_MIBRootName` before calling the `get_ObjectName` functions for that MIB group.

If the value of `MIBRootName_refresh_rate` is set to 0, the refresh function is called every time an object is queried by a management station. In that case, the refresh function is not invoked if no requests are received by the agent for these objects.

### Example:

```
int beaExTable_refresh_rate = 10 /* seconds */;
```

## v\_MIBRootName

This is a structure of type `s_MIBRootName`, declared in the `MIBRootName.h` header file. A structure of this sort is declared for each MIB group or table entry (row).

The agent stores the most recently retrieved values of managed objects in these structures. When a `get_ObjectName` function retrieves a value in response to a GET request from a management station, it retrieves the value from this agent cache.

**Note:** Normally you will not need to modify this structure declaration.

The first field in the structure is always:

```
int index_oid[MAX_INDEX_OID];
```

and the second field in the structure must always be:

```
int oid_len;
```

The structure contains a component:

*v\_MIBRootName.val\_ObjectName*

for each object in the target MIB group or table. For each such component there is another component:

*v\_MIBRootName.sts\_ObjectName*

that contains the status of the object in the cache. The status may be set to either `SNMP_VALID` or `SNMP_INVALID`. If *val\_ObjectName* contains a valid value, the corresponding *sts\_ObjectName* should be set to `SNMP_VALID`. If the value of *ObjectName* cannot be provided, *sts\_ObjectName* must be set to `SNMP_INVALID`.

If *ObjectName* is of type string, there will be a component:

*v\_MIBRootName.len\_ObjectName*

which specifies the length of the string in bytes. Each component in a table row structure corresponding to the MIB INDEX clause must have a valid value if table-handing is to work correctly. Because the INDEX specifies an instance of the table entry (row) object, the value of INDEX affects the OID of all the objects in the row.

The following is an example declaration of a structure for a MIB table object, `beaExTable`:

```
struct s_beaExTable {
oid index_oid[MAX_INDEX_OID]; /* must be the first element */
int oid_len; /* must be the second element */
int row_status; /* only for internal use */
int val_beaExTblIndex;
int sts_beaExTblIndex;
int val_beaExTblIntrO;
int sts_beaExTblIntrO;
int val_beaExTblIntrW;
int sts_beaExTblIntrW;
char val_beaExTblStrRO[MAX_beaExTblStrRO];
int len_beaExTblSrRO;
int sts_beaExTblStrRO;
int val_beaExTblStrRW[MAX_beaTblStrRW];
int len_beaExTblStrRW;
int sts_beaExTblStrRW;
};
extern struct s_beaExTable *v_beaExTable;
```

The constants `MAX_beaExTblStrRO` and `MAX_beaExTblStrRW` define the length, in bytes, of string objects. These constants are also defined in the header file.

# Generated Functions

This section describes generated functions that normally do not need to be modified by the programmer.

## get\_ObjectName

These functions retrieve the current value of *ObjectName* from the agent cache. These functions do not directly access the managed resource. Normally it is not necessary to modify these functions.

The syntax of the get functions depends on the type of *ObjectName*.

**Syntax** For columnar objects that are strings or OIDs:

```
int get_ObjectName(row_index, val_ptr, val_len)
int row_index;
u_char **val_ptr;
int *val_len;
```

For columnar objects other than strings or OIDs:

```
int get_ObjectName(row_index, val_ptr)
int row_index;
int *val_ptr;
```

For scalar objects that are OIDs or strings:

```
int get_ObjectName(val_ptr, val_len)
u_char **val_ptr;
int *val_len;
```

For scalar objects other than OIDs or strings:

```
int get_ObjectName(val_ptr)
```

**Return Value**     The function returns `SNMP_VALID` if the appropriate value of the object is made available in the location pointed to by `val_ptr`. Otherwise, the function returns `SNMP_INVALID` and the contents of the location pointed to by `val_ptr` are undefined.

If *ObjectName* is a string or IP address, the length of the object in bytes is pointed to by the input argument `val_len`. If *ObjectName* is an `OID`, `val_len` points to the number of nodes in the absolute `OID` path from root. If *ObjectName* is a columnar object, `row_index` refers to the row number in the table; 0 is the first row.

---

# Index

## Symbols

/etc/services file 4-16

## A

access functions

- fleshing out generated skeletons for 4-4
- layer 2-14
- standalone testing of 4-5

agent

- address in SNMP trap 2-12
- anatomy of 2-14
- core layer 2-14
- data flow in 2-15
- how it works 2-16
- steps in development of 4-2

Agent Development Kit

- directory structure of 2-17
- introduction to 2-1
- tools 3-3
- utilities 3-3

agent development process 4-2

agent hooks layer 2-14

Agent Integrator

*See* Integrator 2-3

agents

- building 4-6
- building executable 4-5
- installing 4-19
- manager/agent model 2-2

running more than one on a single host  
2-4

testing 4-6

testing with SNMP utilities 4-5

using multiple SNMP agents on a host  
4-21

writing 4-6

ASN.1

- syntax checking by code generator 4-2
- use in defining MIBs 4-2

atomic set 5-5

authenticationFailure trap 2-13

## B

bea\_sendtrap 3-36

BEA\_SM\_SNMP\_MIBFILE in testing agent  
4-18

bea\_snmptrap 3-38

beamgr.conf 4-12

build\_agent 3-4, 4-5

building agent executable 4-15

building agents

build\_agent 3-4

building the executable 4-5

*See also* access functions 4-5

building an agent

steps to follow 4-2

---

## C

- code, adding your own 4-8
- coldStart trap 2-13
- columnar objects 2-9
- community name 4-12
- constants
  - DELTA\_TableName\_ENTRIES 5-8
  - INIT\_TableName\_ENTRIES 5-8
  - MAX\_StringObjectName 5-8
  - user-modifiable 5-8
- core libraries 2-18
- creating SNMP trap header
  - See* csam\_trap\_create
- csam\_get\_keyword 3-25
- csam\_trap\_add\_var\_bind 3-31
- csam\_trap\_create 3-28
- csam\_trap\_send 3-34

## D

- DELTA\_TableName\_ENTRIES constant
  - 5-8
- development process 4-2
- development process flowchart 4-7
- directory structure 2-17
  - of Agent Development Kit 2-17
- dot-dot notation 2-10

## E

- egpNeighborLoss trap 2-13
- enterprise OID 2-12
- enterpriseSpecific trap 2-13

## F

- function templates 5-1
- functions
  - get\_ObjectName 5-11
  - init\_MIBRootName 5-2
  - refresh\_MIBRootName 5-2

- set\_ObjectName 5-5
- set\_TableName\_row\_create 5-7
- test\_ObjectName 5-4

## G

- generating code to support MIB 3-6
- generic trap type 2-12
- get\_ObjectName function 5-11

## H

- header files, generated by code generator 4-8

## I

- imibgenall 3-6
  - example of use 4-4
  - role in development process 4-3
  - target MIB components 4-3
  - what it does 4-3
- imibprint 3-9
- indexes, instance of OIDs 2-11
- init\_MIBRootName
  - generated function 5-2
- init\_MIBRootName function
  - fleshing out 4-11
- INIT\_TableName\_ENTRIES constant 5-8
- installing Windows NT 3-10
- instrumentation
  - role in agent development 4-2
  - used by access functions 5-1
  - using to develop access functions 4-9
  - what it is 2-3
- instsrv 3-10
- Integrator
  - functionality of 2-3
  - use with multiple SNMP agents 4-21
- Internet Activities Board 2-7



---

## Internet standards

- RFC 1155 2-2
- RFC 1157 2-2
- RFC 1212 2-2, 2-5
- RFC 1213 2-2
- RFC 1227 2-2, 2-3

## ISO 2-2

## K

### keywords

- defining 4-12
- in beamgr.conf 4-12

## L

- libraries, core 2-18
- linkDown trap 2-13
- linkUp trap 2-13
- logging SNMP trap messages
  - See snmptrapd*

## M

- managed objects 2-3, 2-6
- managed resource 2-2
- Management Information Base
  - See MIB*
- manager, integrating agent with 4-5
- manager/agent model 2-2
- master agents 2-3
- master agents and subagents 2-3
- MAX\_StringObjectName constant 5-8
- MIB
  - defining 4-2
  - defining your own 4-2
  - definition file 3-2
  - modifying 4-20
  - role in system management 2-3
- MIB groups 2-8
  - generating access functions for 4-3

## MIB II 2-9

- what they are 2-8
- mib.txt file 3-2
- MIBRootName.c file 4-4
- MIBRootName.h header file 4-8
- MIBRootName\_refresh\_rate variable 5-9
- MIBRootName\_refresh\_rate, setting value of 4-10
- moving OID tree
  - See snmpwalk*
- my.asn1 file 4-3, 4-8

## O

- object identifiers 2-6
  - See OID*
- OBJECT TYPE macro 3-2
- OID tree 2-6
- OIDs
  - absolute 2-10
  - conversion by Development Kit utilities 3-2
  - cross-referenced lists of 4-16
  - instance indexes 2-11
  - instance indexes for 2-11
  - relative 2-10
  - retrieving values 2-11
  - specifying 2-10
  - what they are 2-6

## P

- PDU
  - See Protocol Data Unit* 2-12
- PDU type 2-12
- performs get on MIB
  - See snmpget*
- performs getnext on MIB
  - See snmpget*
- performs set on MIB
  - See snmpset*

---

## ports

- assigning agents to 4-21
- other than 161 4-21

## printing MIB names

*See imibprint*

## printing MIB objects

*See imibprint*

## programming guidelines 4-12

- adding rows 4-13
- community 4-12
- defining keywords 4-12
- deleting rows 4-13
- sample traps program 4-12

## Protocol Data Unit (PDU) of SNMP traps 2-12

## R

### receiving SNMP trap messages

*See snmptrapd*

### refresh rate

role of 4-10

*See MIBRootName\_refresh\_rate 5-9*

### refresh\_MIBRootName function 5-2

### RELEASE\_ROOT environment variable 4-3

### reporting information about scalar objects 3-11

### returning keyword value

*See csam\_get\_keyword*

### returning next entry

*See snmpgetnext*

### returning next object in MIB

*See snmpgetnext*

### RFC 1155 3-2

### RFC 1157 3-2

### RFC 1212 4-2

### RFC 1213 3-2

### RFC 1227 2-3

### row addition 4-13

### row deletion 4-13

## S

### scalar objects 2-8, 2-9

### sending SNMP trap

bea\_sendtrap 3-36

bea\_snmptrap 3-38

csam\_trap\_send 3-34

snmptrap 3-18

### set functions 4-11

### set\_ObjectName function 5-5

### set\_TableName\_row\_create function 5-7

### sid

provides access to community 4-12

### SMUX 2-3

### SNMP

agents, more than one on a single host 4-21

architecture 2-5

manager, integrating your MIB with 4-5

requests, format of 3-2

### SNMP Multiplex (SMUX) protocol 2-3

### SNMP traps

agent address 2-12

enterprise OID 2-12

generic trap type 2-12

PDU type 2-12

sending 3-34

specific trap type 2-12

variable bindings 2-12

### snmpget 3-11

### snmpgetnext 3-13

### snmpset 3-15

### snmptrap 3-18

### snmptrapd 3-21

### snmpwalk

syntax of 3-22

utility, how to use 4-18

### specific trap type 2-12

### structures, v\_MIBRootName 5-9

### subagents 2-3

---

## T

- tabular objects 2-8
- test functions, using to test validity of set requests 4-11
- test\_ObjectName functions 5-4
  - role in atomic set 5-4
- test\_TableName\_row\_create function
  - functions
    - test\_TableName\_row\_create 5-6
- testing
  - access functions 4-14
  - setting BEA\_SM\_SNMP\_MIBFILE for 4-18
- trap notification 2-2, 2-12
- traps 4-12
  - authenticationFailure 2-13
  - coldStart 2-13
  - egpNeighborLoss 2-13
  - enterpriseSpecific 2-13
  - generic types 2-13
  - linkDown 2-13
  - linkUp 2-13
  - sample program for generating 4-12
  - structure of 2-12
  - warmStart 2-13
- troubleshooting 4-19

## U

- updating code to support MIB 3-6

## V

- v\_MIBRootName
  - assigning values to components 4-9
  - declaration of 5-2
- v\_MIBRootName cache structure 5-9
- val\_len 5-12
- variable bindings 2-12
- variable bindings for SNMP trap, specifying 3-31
- variables
  - MIBRootName\_refresh\_rate 5-9
  - user-modifiable 5-8

## W

- warmStart trap 2-13

