VSI OpenVMS

VSI TCP/IP Services for OpenVMS SNMP Programming and Reference

VSI TCP/IP Services for OpenVMS SNMP Programming and Reference

VMS Software

Preface

The VSI TCP/IP Services for OpenVMS product is the VSI implementation of the TCP/IP networking protocol suite and internet services for OpenVMS Alpha and OpenVMS VAX systems.

A layered software product, TCP/IP Services provides a comprehensive suite of functions and applications that support industry-standard protocols for heterogeneous network communications and resource sharing.

This manual describes the features of the Simple Network Management Protocol (SNMP) provided with TCP/IP Services. It also describes the extensible SNMP (eSNMP) application programming interface (API)and development environment.

See the *VSI TCP/IP Services for OpenVMS Installation and Configuration* manual for information about installing, configuring, and starting this product.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

VSI seeks to continue the legendary development prowess and customer-first priorities that are so closely associated with the OpenVMS operating system and its original author, Digital Equipment Corporation.

2. Intended Audience

This manual is for experienced OpenVMS and UNIX system managers and assumes a working knowledge of TCP/IP networking, TCP/IP terminology, and some familiarity with the TCP/IP Services product.

3. Document Structure

This manual contains the following chapters:

*   Chapter 1 describes the implementation of eSNMP provided with TCP/IP Services.

*   Chapter 2 describes the groups and objects implemented with the Host Resources MIB and MIB II that are provided with the eSNMP software.

*   Chapter 3 describes how to use the eSNMP API to create a MIB subagent to manage entities or applications.

*   Chapter 4 describes the trap sender, trap receiver,and MIB browser utilities provided with TCP/IP Services.

*   Chapter 5 provides reference information about the eSNMP API routines.

*   Chapter 6 describes some troubleshooting aids provided with TCP/IP Services.

4. Related Documents

The table below lists the documents available with this version of TCP/IP Services.

Table 1. TCP/IP Services Documentation

Manual	Contents
VSI TCP/IP Services for OpenVMS Concepts and Planning	This manual provides conceptual information about TCP/IP networking on OpenVMS systems,including general planning issues to consider before configuring your system to use the TCP/IP Services software. This manual also describes the manuals in the documentation set,and provides a glossary of terms and acronyms for the TCP/IP Services software product.
VSI TCP/IP Services for OpenVMS Installation and Configuration	This manual explains how to install and configure the TCP/IP Services product.
VSI TCP/IP Services for OpenVMS User's Guide	This manual describes how to use the applications available with TCP/IP Services such as remote file operations, email, TELNET, TN3270, and network printing.
VSI TCP/IP Services for OpenVMS Management	This manual describes how to configure and manage the TCP/IP Services product.
VSI TCP/IP Services for OpenVMS Management Command Reference	This manual describes the TCP/IP Services management commands.
VSI TCP/IP Services for OpenVMS ONC RPC Programming	This manual presents an overview of high-level programming using open network computing remote procedure calls (ONC RPC). This manual also describes the RPC programming interface and how to use the RPCGEN protocol compiler to create applications.
VSI TCP/IP Services for OpenVMS Sockets API and System Services Programming	This manual describes how to use the Sockets API and OpenVMS system services to develop network applications.
VSI TCP/IP Services for OpenVMS SNMP Programming and Reference	This manual describes the Simple Network Management Protocol (SNMP) and the SNMP application programming interface (eSNMP). It describes the subagents provided with TCP/IP Services, utilities provided for managing subagents, and how to build your own subagents.
VSI TCP/IP Services for OpenVMS Guide to IPv6	This manual describes the IPv6 environment, the roles of systems in this environment, the types and function of the different IPv6addresses, and how to configure TCP/IP Services to access the IPv6 network.

For a comprehensive overview of the TCP/IP protocol suite, refer to the book *Internetworking with TCP/IP:Principles, Protocols, and Architecture*, by Douglas Comer.

5. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: `<docinfo@vmssoftware.com>`. Users who have OpenVMS support contracts through VSI can contact `<support@vmssoftware.com>` for help with this product.

6. Conventions

The following conventions may be used in this manual:

Convention	Meaning
Ctrl/*x*	A sequence such as **Ctrl**/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
[|]	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are options; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type).

Convention	Meaning
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
Monospace type	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Other conventions are:

- All numbers are decimal unless otherwise noted.

- All Ethernet addresses are hexadecimal.

Chapter 1. Overview

The Simple Network Management Protocol (SNMP) is the de facto industry standard for managing TCP/IP networks. The protocol defines the role of a network management station (NMS) and the SNMP agent. SNMP allows remote users on an NMS to monitor and manage network entities such as hosts, routers, X terminals, and terminal servers.

TCP/IP Services provides support for SNMP Version 2, using the Extensible Simple Network Management Protocol (eSNMP) architecture, under which a single master agent can support any number of subagents. The TCP/IP Services implementation of eSNMP includes a master agent, two subagents, an application programming interface (API), tools used to build additional subagents, startup and shutdown procedures, and text-based configuration files.

This chapter provides an overview of the VSI OpenVMS implementation of eSNMP. Topics include:

- eSNMP master agent and subagent architecture (Section 1.1)

- The procedure for handling SNMP requests (Section 1.2)

- The components of the TCP/IP Services software kit that implement SNMP (Section 1.3)

- The files useful in developing your own subagent (Section 1.4)

- The eSNMP API (Section 1.5)

- The management information base (MIB) compiler (Section 1.6)

- The impact of running SNMP Version 1 subagents against the SNMP Version 2 implementation provided with TCP/IP Services (Section 1.7)

- Sources of additional information about implementing subagents (Section 1.8)

1.1. SNMP Architecture

Figure 1.1 illustrates the SNMP architecture.

Figure 1.1. SNMP Architecture
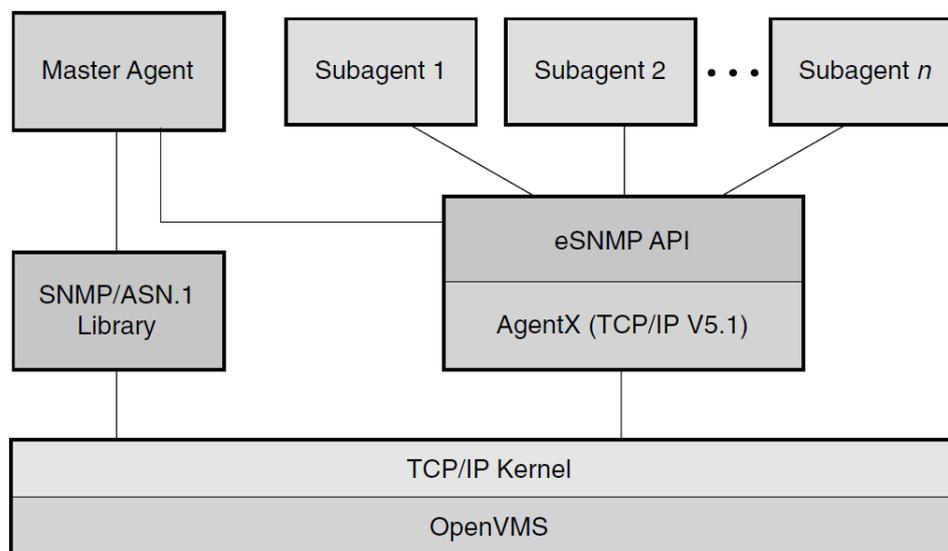
The SNMP environment consists of the following elements:

- The master agent, a process that runs on the host and handles SNMP requests from clients over the standard SNMP well-known port 161.

- One or more subagents, each of which provides access to the MIB data specified in client requests. In the TCP/IP Services implementation, the master agent contains two resident subagents, one that handles a subset of MIB II variables, and another that handles the Host Resources MIB. These MIBs are described in Chapter 2.

- The SNMP ASN.1 library, used by the master agent to interpret ASN.1 messages.

- The eSNMP API, the application programming interface that provides routines for programming your own subagents. This API runs on the AgentX routines, which are internal to the SNMP architecture.

- The TCP/IP kernel running on the OpenVMS operating system.

  The master agent and subagents communicate by means of the AgentX protocol, which is based on RFC 2741.

For information about configuring and managing the SNMP service, refer to the *VSI TCP/IP Services for OpenVMS Management* guide.

1.2. Request Handling

The eSNMP software manages network communication by having the master agent listen for requests and then passes the requests to the appropriate subagent.

Figure 1.2 illustrates communication between the master agent and subagents.

Figure 1.2. eSNMP Data Flow
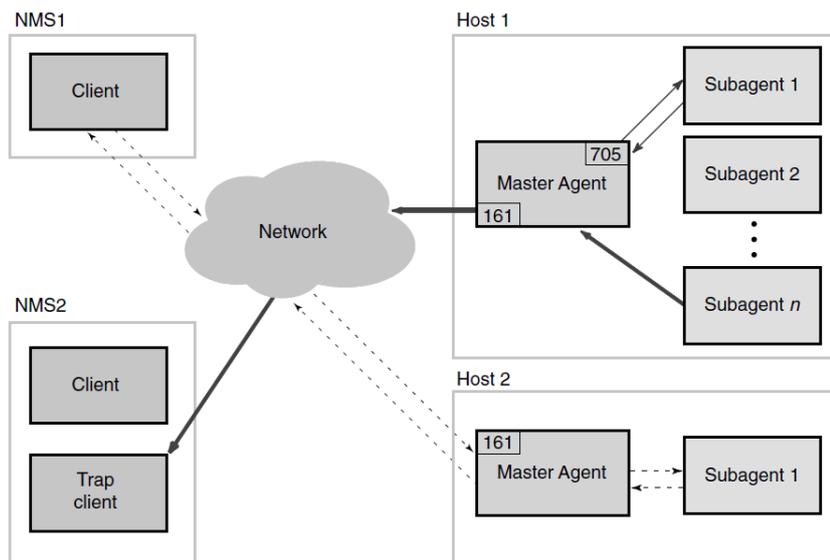
The process of communication for a request is illustrated with dashed lines and includes the following steps:

1. The network management station (NMS) (sometimes called the client), originates SNMP requests to obtain and set information.

Note

The client component is not provided with TCP/IP Services.

To provide access to MIBs and to test SNMP communication, TCP/IP Services provides the following utilities:

- MIB browser

- Trap sender

- Trap receiver

These utilities are described in Chapter 4.

The network management station sends an SNMP request to the master agent running on the host, using port 161. An SNMP request is made using one of the following commands:

- `Get`

- `GetNext`

- `GetBulk`

- `Set`

Note

TCP/IP Services does not support the standard SNMP `Inform` command.

The request specifies the object identifier (OID) of the data to be accessed. For information about formatting `get` and `set` requests, refer to Section 5.2. Request formats are specified in RFC 1905.

2. The master agent sends the request to the subagent that registered the subtree containing the OID.

The subagent receives communications from the master agent over the socket that was assigned when the subagent registered the subtree.

3. The appropriate subagent processes the request.

4. The subagent sends the response message to the master agent using the port that was assigned when the subagent registered the MIB.

When they are idle, subagents periodically send a message to port 705 to ensure that the master agent is still running. In Figure 1.2, subagent 1 is sending the `esnmp_are_you_there` message.

A `trap` is generated by the subagent and sent to the client. In Figure 1.2, subagent n is generating a trap for the trap client on NMS 2.

The `trap` and `esnmp_are_you_there` routines are described in Section 5.1.

1.3. TCP/IP Services Components for SNMP

Table 1.1 lists the components of SNMP and the command procedures for managing SNMP that are supplied with TCP/IP Services.

Table 1.1. SNMP Component Files

File	Location	Function
TCPIP$ESNMP_SERVER.EXE	SYS$SYSTEM	Master agent image.
TCPIP$OS_MIBS.EXE	SYS$SYSTEM	MIB II subagent image.
TCPIP$HR_MIB.EXE	SYS$SYSTEM	Host Resources MIB subagent image.
TCPIP$SNMP_REQUEST.EXE	SYS$SYSTEM	Simple MIB browser.
TCPIP$SNMP_TRAPSND.EXE	SYS$SYSTEM	Utility for sending trap messages.
TCPIP$SNMP_TRAPRCV.EXE	SYS$SYSTEM	Utility for receiving trap messages.
TCPIP$ESNMP_SHR.EXE	SYS$SHARE	Image file containing eSNMP application programming interface (API) routines.
TCPIP$SNMP_STARTUP.COM	SYS$STARTUP	Command procedure that installs master and subagent images and runs TCPIP$SNMP_RUN.COM.
TCPIP$SNMP_SYSTARTUP.COM	SYS$STARTUP	Command procedure initiated by TCPIP$SNMP_STARTUP.COM. Provided for site-specific customizations, such as parameter settings.
TCPIP$SNMP_RUN.COM	TCPIP$SYSTEM	Command procedure that starts the master agent and subagents.
TCPIP$SNMP_SHUTDOWN.COM	SYS$STARTUP	Command procedure that stops the master agent and subagents.
TCPIP$SNMP_SYSHUTDOWN.COM	SYS$STARTUP	Command procedure initiated by TCPIP$SNMP_SHUTDOWN.COM. Provided for site-specific customization, such as parameter settings.
TCPIP$EXTENSION_MIB_STARTUP.COM	SYS$SYSDEVICE:[TCPIP$SNMP]	
		Command procedure invoked by TCPIP$SNMP_SYSTARTUP.COM to start custom subagents.

File	Location	Function
TCPIP$EXTENSION_MIB _SHUTDOWN.COM	SYS$SYSDEVICE:[TCPIP$SNMP]	
		Command procedure invoked by TCPIP$SNMP_SYSHUTDOWN.COM to stop custom subagents.
TCPIP$EXTENSION _MIB_RUN.COM	SYS$SYSDEVICE:[TCPIP$SNMP]	
		Command procedure invoked by TCPIP$SNMP_SYSTARTUP.COM when the service is enabled and starts detached processes to run subagents.

1.4. Writing an eSNMP Subagent

Table 1.2 lists the files that are available to help you develop MIBs and subagents. Except where noted, the files are located in the directory pointed to by TCPIP$SNMP_EXAMPLES.

Table 1.2. Files for Building a Subagent

File	Description
ESNMP.H	Header file used to create a subagent. Located in TCPIP$ESNMP.
GAWK.EXE	Interpreter for MIB converter.
MIB-CONVERTER.AWK	A UNIX based awk shell script that takes a MIB definition in ASN.1 notation and converts it to an .MY file.
RFC1213.MY	MIB II definitions.
RFC1231.MY	IEEE 802.5 Token Ring MIB definitions.
RFC1285.MY	FDDI MIB definitions.
RFC1442.MY	SNMP Version 2 Structure of Management Information (SMI) definitions.
SNMP-SMI.MY	SNMP Version 2 SMI definitions from RFC 1902 (replaces RFC 1442).
SNMP-TC.MY	SNMP Version 2 SMI definitions from RFC 1903 (replaces RFC 1443).
V2-TC.MY	SNMP Version 2 SMI definitions from RFC 1903 (superset of those in SNMP-TC.MY).
TCPIP$BUILD_CHESS.COM	Command file that builds the sample chess subagent.
TCPIP$CHESS_SUBAGENT.OPT	Options file for use in building the sample chess subagent.

File	Description
*.C and *.H	Source code for chess example. Contains detailed documentation that explains how the code functions.
TCPIP$CHESS_SUBAGENT.EXE	Functioning chess example image.
TCPIP$ESNMP.OLB	Object library file containing routines used to create a subagent. Located in the directory pointed to by TCPIP$SNMP.
TCPIP$ESNMP_SHR.EXE	Shareable image containing routines used to create a subagent. Located in the directory pointed to by SYS$SHARE.
UCX$ESNMP_SHR.EXE	Copy of TCPIP$ESNMP_SHR.EXE, provided for compatibility with existing customer subagents linked under TCP/IP Services V4.x. Located in the directory pointed to by SYS$SHARE.
TCPIP$MIBCOMP.EXE TCPIP$MOSY.EXE TCPIP$SNMPI.EXE	Images associated with the MIB compiler. Located in SYS$SYSTEM.

For information about building a subagent on an OpenVMS system, see Chapter 3.

1.5. The eSNMP API

The TCP/IP Services implementation of the eSNMP architecture includes an API that provides programmers with many eSNMP routines they would otherwise have to develop themselves.

The eSNMP API includes interface routines, method routines, and support routines.

Interface routines handle the basic subagent operations, such as:

- Subagent initialization and termination

- Registration

- Polling of the master agent

- Trap sending

- UNIX system time conversion

- Adding and removing subagent capabilities registered with the master agent

The support routines allow the subagent to manipulate the data in the response to the request, and include the following:

- Basic protocol data unit (PDU) handling

- Authentication handling

- Octet string handling

- Variable binding (VARBIND) handling

- Object identifier (OID) handling

- Buffer handling

Chapter 5 describes the API routines in more detail.

To create a subagent, the programmer must provide modules to implement the method routines, as described in Chapter 3.

1.5.1. The SNMP Utilities

To provide quick access to information in the MIBs, and to test SNMP operation, TCP/IP Services provides the following utilities:

- TCPIP$SNMP_REQUEST.EXE, a MIB browser that allows you to retrieve and update objects from the MIBs.

- TCPIP$SNMP_TRPSND.EXE, a trap sender that generates traps (messages that require no response).

- TCPIP$SNMP_TRPRCV.EXE, a trap receiver (or "listener") that is used to detect trap messages.

For information about using the SNMP utilities, see Chapter 4.

1.6. The MIB Compiler

The MIB compiler processes the statements in an ASN.1 file and generates modules that are used by the developer to create subagent routines. For every ASN.1 input file that is processed using the MIB compiler, two output files, a *subtree*_TBL.H file and a *subtree*_TBL.C file, are generated, where *subtree* is the name from the original MIB definition file (for example, chess). The output files are described in more detail in Chapter 3.

The *subtree*_TBL.H file is a header file that contains the following:

- A declaration of the subtree structure

- Index definitions for each MIB variable in the subtree

- Enumeration definitions for MIB variables with enumerated values

- MIB group data structure definitions

- Method routine function prototypes

The *subtree*_TBL.C file is an object file that contains the following:

- An array of integers representing the OIDs for each MIB variable

- An array of OBJECT structures

- An initialized SUBTREE structure

1.7. SNMP Versions

The extensible SNMP software supports SNMP Version 2, based on RFCs 1901 through 1908, including:

- The SNMP Version 2 structure of management information for SNMP Version 2 (SMI Version 2) and textual conventions.

- The eSNMP library API (SNMP Version 2), variable binding exceptions, and error codes.

- SNMP Version 1 and SNMP Version 2 requests. Both versions are handled by the master agent. SNMP Version 2 specific information from the subagent is mapped, when necessary, to SNMP Version 1 adherent data (according to RFC 2089). For example, if a management application makes a request using SNMP Version 1 PDUs, the master agent replies using SNMP Version 1 PDUs, mapping any SNMP Version 2 SMI items received from subagents. In most cases, subagents created with a previous version of the eSNMP API do not require any code changes and do not have to be recompiled. The circumstances under which recoding or recompiling are required are described in Section 1.7.1.

1.7.1. Using Existing (SNMP Version 1) MIB Modules

Existing SNMP Version 1 MIB subagent executable files should be compatible with the current SNMP Version 2 master agent without the need to recompile and relink, with the following exceptions:

- Any program that relies on TCP/IP Services Version 4.1 or 4.2 kernel data structures or access functions may run but may not return valid data. Such programs should be rewritten.

- Programs linked against UCX$ACCESS_SHR.EXE, UCX$IPC_SHR.EXE, or other older shareable images (except for UCX$ESNMP_SHR.EXE, which is described in the next list item) may not run even when relinked. You may need to either rewrite or both rewrite and recompile such programs. Note that the Chess example image (UCX$CHESS_SUBAGENT.EXE) has been updated and renamed TCPIP$CHESS_SUBAGENT.EXE.

- For programs linked against certain versions of UCX$ESNMP_SHR.EXE:

  - Images associated with the following versions of TCP/IP Services will run correctly without the need to relink them:

    Version 4.1 ECO 9 and later
    Version 4.2 ECO 1 and later

    The installation of TCP/IP Services provides a backward-compatible version of UCX$ESNMP_SHR.EXE in the SYS$SHARE directory. Do not delete this image.

    If you have problems running images linked against an older version of UCX$ESNMP_SHR.EXE, verify that the version in SYS$SHARE is the latest by entering the following DCL command:

    ```
    $ DIRECTORY/DATE SYS$SHARE:*$ESNMP_SHR.EXE
    ```

    The creation dates of the files with the prefix TCPIP$ and UCX$ should be within a few seconds of each other, and only one version of each file should exist. Make sure both images include the file protection W:RE.

- You should relink and perhaps recompile images associated with ECOs for Version 4.1 or 4.2 other than those discussed in the previous list item.

Images linked against object library (.OLB) files may not need to be relinked, although you can relink them against the shareable images in this version of the product to decrease the image size. Relinking against the shareable image allows you to take advantage of updated versions of the eSNMP API without the need to relink. Some images linked against the current version of TCP/IP Services may run under Versions 4.1 and 4.2, but this backward compatibility is not supported and may not work in future versions of TCP/IP Services.

If an existing subagent does not execute properly, relink it against this version of TCP/IP Services to produce a working image. Some subagents (such as the OpenVMS Server MIB) require a minimum version of OpenVMS as well as a minimum version of TCP/IP Services.

1.8. For More Information

This manual provides the OpenVMS information required for implementing eSNMP subagents and ensuring their proper operation in that environment.

For information about prototypes and definitions for the routines in the eSNMP API, see the TCPIP $SNMP:ESNMP.H file.

Table 1.2 lists files that contain additional comments and documentation.

Chapter 2. MIBs Provided with TCP/IP Services

This chapter describes how MIBs are implemented on OpenVMS. The MIBs provided with TCP/IP Services are:

- The Host Resources MIB, which manages operating system objects (Section 2.1)

- MIB II, which manages TCP/IP kernel objects (Section 2.2)

2.1. Overview of the Host Resources MIB

The Host Resources MIB defines a uniform set of objects useful for the management of host computers. The Host Resources MIB, described by RFC 1514, defines objects that are common across many computer system architectures. The TCP/IP Services implementation of SNMP includes many of these defined objects. In addition, some objects in MIB II provide host management functionality.

2.1.1. Defining Host Resources MIB Implemented Objects

This section defines each of the implemented eSNMP objects. Table 2.1 provides a general RFC description and a specific OpenVMS description for each implemented object.

Table 2.1. Host Resources MIB Objects

Object Name	RFC Description	OpenVMS Description
hrSystemUptime	The amount of time since this host was last initialized.	Time since system boot (in hundredths of a second).
hrSystemDate	The host's notion of the local date and time of day.	Date and time character string with Coordinated Universal Time (UTC) information if available.
hrSystemIntialLoadDevice	Index of the hrDeviceEntry for configured initial operating system load.	Index of SYS$SYSDEVICE in the device table.
hrSystemIntialLoadParameters	Parameters supplied to the load device when requesting initial operating system configuration.	A string of boot parameters from the console (Alpha only).
hrSystemNumUsers	Number of user sessions for which the host is storing state information.	Number of processes that are neither owned by another process nor running detached.
hrSystemProcesses	Number of process contexts currently loaded or running on the system.	Number of processes listed using the SHOW SYSTEM command.
hrSystemMaxProcesses	Maximum number of process contexts the system can support, or 0 if not applicable.	SYSGEN parameter MAXPROCESSCNT.

Object Name	RFC Description	OpenVMS Description
hrMemorySize	The amount of physical main memory contained in the host.	The amount of physical main memory contained in the host.
hrStorageIndex	A unique value for each logical storage area contained in the host.	Index of entry in hrStorageTable.
hrStorageType	The type of storage represented by this entry.	A numeric representation of the device class and type displayed by the SHOW DEVICE/FULL command.
hrStorageDescr	A description of the type and instance of the storage described by this entry.	Character string device type displayed by the SHOW DEVICE/FULL command.
hrStorageAllocationUnits	The size of the data objects allocated from this pool (in bytes).	Always 512 (the size of an OpenVMS disk block).
hrStorageSize	The size of storage in this entry in hrStorageAllocationUnits.	The total number of blocks on a device displayed by the SHOW DEVICE/FULL command.
hrStorageUsed	The allocated amount of storage in this entry inhrStorageAllocationUnits.	The total number of used blocks on a device displayed by the SHOW DEVICE/FULL command.
hrDeviceIndex	A unique value for each host or device constant between agent reinitialization.	Index of entry in hrDeviceTable.
hrDeviceType	An indication of the type of device. Some of these devices have corresponding entries in other tables.	In object identifier format, a numeric representation of the device class and type displayed by the SHOW DEVICE/FULL command.
hrDeviceDesc	A text description of the device, including manufacturer and version number (service, optional).	Character string of the device type displayed by the SHOW DEVICE/FULL command.
hrDeviceStatus	The current operational state of the device.	A numeric indication of the status of the device.
hrDeviceErrors	The number of errors detected on the device. The recommended initial value is zero.	The number of errors indicated by the SHOW DEVICE command. This value is initialized to zero when the device is recognized by the system instead of when the master agent is initialized.
hrProcessorFrwID	The product ID of the firmware associated with the processor.	An object identifier that corresponds to the console or PALcode version (Alpha only).

Object Name	RFC Description	OpenVMS Description
hrNetworkIfIndex	The value of the if Index that corresponds to this network device.	The value of the index in the interface table in the standard MIB that corresponds to this network device.
hrDiskStorageAccess	Indicates whether the storage device is read/write or read only.	This value is set to 2 if the device is read only; otherwise, it is set to 1. (The SHOW DEVICE/FULL command displays "software write-locked.")
hrDiskStorageMedia	Indicates the storage device media type.	Indicates device type.
hrDiskStorageRemovable	Indicates whether the disk can be removed from the drive.	Indicates whether the disk can be removed from the drive.
hrDiskStorageCapacity	The total size of this long-term storage device.	Half of the value for total blocks displayed by the SHOW DEVICE/FULL command.
hrSWRunIndex	A unique value for each software product running on the host.	Process ID.
hrSWRunPath	A description of the location where this software was loaded.	Fully qualified name of executable image.
hrSWRunStatus	The status of the software that is running.	The values and the associated status of each are: • 1 indicates that the current process is running (CUR) • 2 indicates that the process is computable (COM) • 3 indicates that you cannot run the process.
hrSWRunPerfCPU	The number (in hundredths of a second) of the total system's CPU resources consumed by this process.	Process elapsed CPU time (in hundredths of a second).
hrSWRunPerfMem	The total amount of real system memory allocated to this process.	Process current working set (in kilobytes).

2.1.2. Restrictions to Host Resources MIB

SNMP requests are not implemented for the following Host Resources MIB objects:

hrPartitionTablehrPrinterTablehrSWInstalledhrSWInstalledTable

SNMP set requests are not implemented for the following Host Resources MIB objects:

hrFSLastFullBackupDate

```
hrFSLastPartialBackupDate
hrStorageSize
hrSWRunStatus
hrSystemDate
hrSystemInitialLoadDevice
hrSystemInitialLoadParameters
```

Note

For objects that are not implemented, the Host Resources MIB returns a `NoSuchObject` error status.

TCP/IP Services supports the objects in the Host Resources MIB as follows:

- The `hrDeviceTable` includes all the devices known to the OpenVMS host except those with the following characteristics:

    - Off line

    - Remote

    - UCB marked delete-on-zero-reference-count

    - Mailbox device

    - Device with remote terminal (DEV$M_RTT characteristic)

    - Template terminal-class device

    - LAT device (begins with _LT)

    - Virtual terminal device (begins with _VT)

    - Pseudoterminal device (begins with _FT)

    Data items in the `hrDeviceTable` group have the following restrictions:

    - `hrDeviceID` is always null OID (0.0).

    - `hrDeviceErrors` is coded as follows:

Code	Condition
warning (3)	Error logging is in progress (OpenVMS UCB value UCB$M_ERLOGIP).
running (2)	Software is valid and no error logging is in progress (OpenVMS UCB value UCB$M_VALID).
unknown (1)	Any other OpenVMS status.

    The `hrDeviceTable` now includes template devices (for example, DNFS0 for NFS and DAD0 for virtual devices).

For network devices, only the template devices (those with unit number 0) are displayed.

- `hrFSMountPoint` (1.3.6.1.2.1.25.3.8.1.2) is DNFS *n*. The device may change between restarts or after a dismount/mount procedure.

- In the `hrFSTable` group, if no file systems are mounted through NFS or no information is accessible, a `"no such instance"` status is returned for a `get` request. Browsers respond differently to this message. For example, TCPIP$SNMP_REQUEST.EXE responds with no output and returns directly to the DCL prompt.

  After an NFS mount, the following information is returned in response to a `Get` request. The data items implemented for OpenVMS (refer to RFC 1514) are:

  - `hrFSIndex`.

  - `hrFSMountPoint` is the local DNFS device name.

  - `hrFSRemoteMountPoint` is the remote file system.

  - `hrFSType` is implemented as:

    - OID 1.3.6.1.2.1.25.3.9.1, for OpenVMS if the file system is not a UNIX style container file system.

    - `hrFSNFS`, OID 1.3.6.1.2.1.25.3.9.14, if the file system is a TCP/IP Services container file system or a UNIX host.

  - `hrFSAccess`, as defined in RFC 1514.

  - `hrFSBootable` is always HRM_FALSE (integer 2).

  - `hrFSStorageIndex` is always 0.

  - `hrFSLastFullBackupDate` is unknown time. This entry is encoded according to RFC 1514 as a hexadecimal value 00-00-01-01-00-00-00-00 (January 1, 0000).

  - `hrFSLastPartialBackupDate` is unknown time. This information is not available for OpenVMS systems. Instead, hexadecimal value 00-00-01-01-00-00-00-00 (January 1, 0000) applies.

- `hrProcessorFrwID` (OID prefix 1.3.6.1.2.1.25.3.3.1.1) is not implemented on OpenVMS VAX. On this type of system, it returns standard null OID (0.0). For example:

  ```
  1.3.6.1.2.1.25.3.3.1.1.1 = 0.0
  ```

  For OpenVMS Alpha (firmware version 5.56-7), the response is shown in the following example:

  ```
  1.3.6.1.2.1.25.3.3.1.1.1 = 1.3.6.1.2.1.25.3.3.1.1.1.5.56.7
  ```

- Data items in the `hrDiskStorage` table have the following restrictions:

  - `hrDiskStorageMedia` is always "unknown" (2).

  - `hrDiskStorageRemoveble` is always "false" (2). Note the incorrect spelling of "removable" in `hrDiskStorageRemoveble` (from RFC 1514).

- `hrStorageType` always contains the value of `hrStorageFixedDisk` (1.3.6.1.2.1.25.2.1.4).

2.2. Overview of MIB II

The Standard MIB (MIB II) described in RFC 1213 defines a set of objects useful for managing TCP/IP Internet entities. MIB II supports network monitoring and managing from the Transport layer down to the Physical layer of the TCP/IP internet stack. This MIB also provides information on how connections are established and how packets are routed through the Internet. For more information about MIB architecture, see Section 3.2.

2.2.1. MIB II Implemented Groups

A **group** is a subdivision of a MIB that defines a subtree. SNMP as implemented by TCP/IP Services supports the following groups:

- `system (1)`

- `interfaces (2)`

- `Internet Protocol (4)`

- `ICMP (5)`

- `TCP (6)`

- `UDP (7)`

- `SNMP (11)`

  In the SNMP group (1.3.6.1.2.1.11), data elements with the status noted as obsolete in RFC 1907 are not implemented.

Note

The TCP/IP Services implementation of SNMP does not support the following defined MIB II groups:

- `at` (address translation) group

- `EGP` (External Gateway Protocol) group

- `transmission` group

2.2.2. Restrictions to MIB II Implementation

SNMP requests are not implemented for the following MIB II objects:

```
ipRouteMetric1 - ipRouteMetric5tcpMaxConn
```

SNMP `set` requests are not implemented for the following MIB II group objects:

```
ipDefaultTTL
ipRouteAge
ipRouteDest
ipRouteIfIndex
ipRouteMaskipRouteNextHop
```

```
ipRouteType
```

The TCP/IP Services implementation of SNMP includes the following MIB II objects:

- `sysObjectID` is returned in the following format:

  ```
  1.3.6.1.2.1.1.2.0 = 1.3.6.1.4.1.36.2.15.13.22.1
  ```

  where 1.3.6.1.4.1.36.2.15.13.22.1 corresponds to:

  ```
  iso.org.dod.internet.private.enterprises.dec.ema.SysObjectIds.DEC-
  OpenVMS.eSNMP
  ```

- The `sysORTable` elements are under OID prefix 1.3.6.1.2.1.1.9.1. See RFC 1907 for details.

  When both the TCPIP$OS_MIBS and TCPIP$HR_MIB subagents are running, a `get` request on the `sysORTable` is as follows. Except where noted, the OIDs conform to RFC 1907.

  ```
  1.3.6.1.2.1.1.9.1.2.1 = 1.3.6.1.4.1.36.15.3.3.1.1
  1.3.6.1.2.1.1.9.1.2.2 = 1.3.6.1.4.1.36.15.3.3.1.2
  1.3.6.1.2.1.1.9.1.3.1 = Base o/s agent (OS_MIBS) capabilities
  1.3.6.1.2.1.1.9.1.3.2 = Base o/s agent (HR_MIB) capabilities
  1.3.6.1.2.1.1.9.1.4.1 = 31 = 0 d 0:0:0
  1.3.6.1.2.1.1.9.1.4.2 = 36 = 0 d 0:0:0
  ```

  This example is from the MIB browser (TCPIP$SNMP_REQUEST.EXE).

- Under certain conditions, a subagent makes a duplicate entry in `sysORTable` when it restarts. For example:

  ```
  1.3.6.1.2.1.1.9.1.2.1 = 1.3.6.1.4.1.36.15.3.3.1.1
  1.3.6.1.2.1.1.9.1.2.2 = 1.3.6.1.4.1.36.15.3.3.1.2
  1.3.6.1.2.1.1.9.1.2.1 = Base o/s agent (OS_MIBS) capabilities
  1.3.6.1.2.1.1.9.1.2.2 = Base o/s agent (OS_MIBS) capabilities
  1.3.6.1.2.1.1.9.1.4.1 = 3256 = 0 d 0:0:32
  1.3.6.1.2.1.1.9.1.4.2 = 3256 = 0 d 0:0:32
  ```

  In this example, the TCPIP$OS_MIBS subagent made two entries with different ID numbers (OIDs with the prefix 1.3.6.1.2.1.1.9.1.2) that may show different `sysORUpTime` (1.3.6.1.2.1.1.9.1.4). The `snmp_request` program translates the value received (in hundredths of a second) to the following, dropping any fractions of seconds:

  ```
  d n hh:mm:ss
  ```

  In this format, *n* represents the number of days, *hh* represents the number of hours, *mm* represents the number of minutes, and *ss* represents the number of seconds.

  The HR_MIB subagent has not yet successfully started and registered its capabilities. If it starts, its entries in this example would use the next available index number.

- On systems running versions of the operating system prior to OpenVMS 7.1-2, counters for the MIB II `ifTable` do not wrap back to 9 after reaching the maximum value ($2^{32} - 1$), as defined in RFC 1155. Instead, they behave like the gauge type and remain at the maximum value until cleared by an external event, such as a system reboot. The following counters are affected:

  ```
  ifInDiscardsifInErrors
  ifInNUcastPkts
  ifInOctets
  ```

```
ifInUcastPkts
ifInUnknownProtos
ifOutErrors
ifOutNUcastPkts
ifOutOctets
ifOutUcastPkts
```

Note that for SNMP Version 2, these counters are data type Counter32. The following `ifTable` members are always -1 for OpenVMS:

`ifOutDiscards` (Counter32)
`ifOutQLen` (Gauge32)

Chapter 3. Creating a Subagent Using the eSNMP API

This chapter describes how to use the eSNMP API to create a MIB subagent that manages entities or applications. Topics included in this chapter are:

- Creating a MIB specification (Section 3.1)

- The structure of management information (Section 3.2)

- Creating a MIB source file (Section 3.3)

- Including the routines and building the subagent (Section 3.4)

- Including your subagents in startup and shutdown procedures (Section 3.5)

Note

To use this eSNMP API to create a subagent, you must have a C compiler running in your development environment.

3.1. Creating a MIB Specification

The creation of a management information base (MIB) begins with data gathering. During this phase, the developer identifies the information to manage, based on the entities that the network manager needs to examine and manipulate. Each resource that a MIB manages is represented by an object. After gathering the data, the developer uses Abstract Syntax Notation 1 (ASN.1) to specify the objects in the MIB.

3.2. The Structure of Management Information

The structure of management information (SMI), which is specified in RFCs 1155 and 1902, describes the general framework within which a MIB can be defined and constructed. The SMI framework identifies the data types that can be used in the MIB and how resources within the MIB are represented and named.

SMI avoids complex data types to simplify the task of implementation and to enhance interoperability. To provide a standard representation of management information, the SMI specifies standard techniques for the following:

- Defining the structure of a particular MIB

- Defining individual objects, including the syntax and value of each object

- Encoding object values

3.2.1. Assigning Object Identification Codes

Each object in a MIB is associated with an identifier of the ASN.1 type, called an object identifier (OID). OIDs are unique integers that follow a hierarchical naming convention.

Each OID has two parts:

- A preassigned portion that is always represented on the SMI tree as 1.3.6.1 or iso (1), org (3), dod (6), Internet (1).

- A developer-assigned portion for the private development of MIBs.

Note

Your organization may require you to register all newly assigned OIDs.

In addition to an OID, you should also assign a name to each object to help with human interpretation.

3.2.2. MIB Subtrees

Understanding MIB subtrees is crucial to understanding the eSNMP API and how your subagent will work.

Note

This manual assumes that you understand the OID naming structure used in SNMP. If not, refer to RFC 1902: Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMP Version 2).

The information in SNMP is structured hierarchically like an inverted tree. Each node has a name and a number. Each node can also be identified by an OID, which is a concatenation of the subidentifiers (non-negative numbers). These numbers are on the path from the root node down to that node in the tree. In this hierarchy, data is associated only with leaf nodes. (A **leaf node** represents a MIB variable that can have an instance and an associated value.)

An OID must be at least two subidentifiers and at most 128 subidentifiers in length. The subidentifier ranges are:

- Subidentifier 1 values range from 0 to 2, inclusive.

- Subidentifier 2 values range from 0 to 39, inclusive.

- The remaining subidentifier values can be any non-negative number.

Figure 3.1 illustrates the SMI hierarchical tree arrangement as specified in RFCs 1155 and 1902.

Figure 3.1. MIB II in SMI Tree Structure
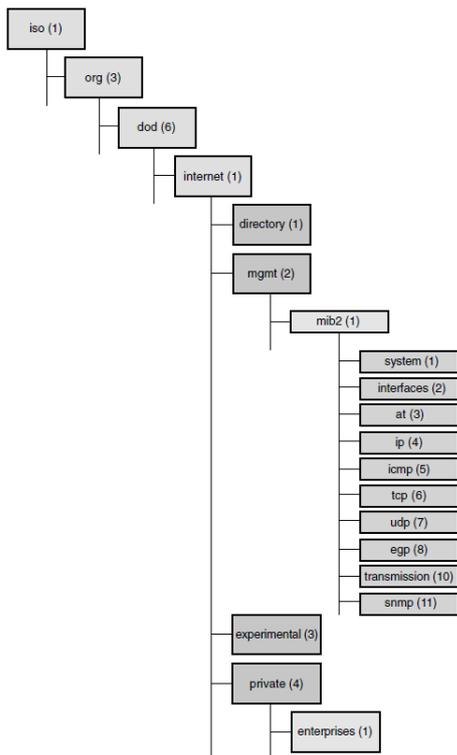
For example, the chess MIB provided with the sample code in the [TCPIP$EXAMPLES.SNMP] directory has an element with the name "chess." The OID for the element chess is 1.3.6.1.4.1.36.2.15.2.99, which is derived from its position in the hierarchy of the tree:

```
iso(1)
 org(3)
  dod(6)
   internet(1)
    private(4)
     enterprise(1)
      digital(36)
       ema(2)
        sysobjects(15)
         decosf(2)
          chess(99)
```

Any node in the MIB hierarchy can define a MIB subtree. All elements in the subtree have an OID that starts with the OID of the subtree base. For example, if you define chess to be a MIB subtree base, the elements with the same prefix as the chess OID are all in the MIB subtree:

```
chess              1.3.6.1.4.1.36.2.15.2.99
 chessProductID    1.3.6.1.4.1.36.2.15.2.99.1
 chessMaxGames     1.3.6.1.4.1.36.2.15.2.99.2
 chessNumGames     1.3.6.1.4.1.36.2.15.2.99.3
 gameTable         1.3.6.1.4.1.36.2.15.2.99.4
  gameEntry        1.3.6.1.4.1.36.2.15.2.99.4.1
   gameIndex       1.3.6.1.4.1.36.2.15.2.99.4.1.1
   gameDescr       1.3.6.1.4.1.36.2.15.2.99.4.1.2
gameNumMoves       1.3.6.1.4.1.36.2.15.2.99.4.1.3
   gameStatus      1.3.6.1.4.1.36.2.15.2.99.4.1.4
 moveTable         1.3.6.1.4.1.36.2.15.2.99.5
```

```
  moveEntry              1.3.6.1.4.1.36.2.15.2.99.5.1
    moveIndex            1.3.6.1.4.1.36.2.15.2.99.5.1.1
    moveByWhite          1.3.6.1.4.1.36.2.15.2.99.5.1.2
    moveByBlack          1.3.6.1.4.1.36.2.15.2.99.5.1.3
    moveStatus           1.3.6.1.4.1.36.2.15.2.99.5.1.4
 chessTraps              1.3.6.1.4.1.36.2.15.2.99.6
  moveTrap               1.3.6.1.4.1.36.2.15.2.99.6.1
```

The base of this MIB subtree is registered with the master agent to tell it that this subagent handles all requests related to the elements in the subtree.

The master agent expects a subagent to handle all objects subordinate to the registered MIB subtree. This principle guides your choice of MIB subtrees. For example, registering a subtree of chess is reasonable because it is realistic to assume that the subagent could handle all requests for elements in this subtree. Registering an entire application-specific MIB usually makes sense because the particular application expects to handle all objects defined in the application-specific MIB.

However, registering a subtree of SNMP (under MIB II) would be a mistake, because it is unlikely that the subagent is prepared to handle every defined MIB object subordinate to SNMP (packet counts, errors, trapping, and so on).

A subagent can register as many MIB subtrees as it wants. It can register OIDs that overlap with other registrations by itself or with other subagents; however, it cannot register the same OID more than once. Subagents can register and unregister MIB subtrees at any time after communication with the master agent is established.

Normally, it is the nonleaf nodes that are registered as a subtree with the master agent. However, leaf nodes, or even specific instances, can be registered as a subtree.

The master agent delivers requests to the subagent that has the MIB subtree with the longest prefix and the highest priority.

3.3. Creating a MIB Source File

Creating the MIB source file requires the following four-step process:

1. Write the ASN.1 input files, as described in Section 3.3.1.

2. Process the input files with the MIB compiler, as described in Section 3.3.2.

3. Compile and link the routines, as described in Section 3.4.

4. Include the subagent, as described in Section 3.5.

3.3.1. Writing the ASN.1 Input File

After you have assigned names and OIDs to all of the objects in the MIB, create an ASN.1 source file using ASCII statements.

Note

Providing information about ASN.1 syntax and programming is beyond the scope of this guide. For more information about ASN.1 programming, refer to one or more of the documents on this subject provided by the International Organization for Standardization (ISO).

Instead of creating ASN.1 files yourself, you can create .MY files from existing ASCII files (for example, from RFCs) by using the MIB-converter facility provided with TCP/IP Services. This facility uses a UNIX `awk` script, which runs on OpenVMS as well as on appropriately configured UNIX systems. For details about the facility, see the MIB-CONVERTER.AWK file, which is located in the [.SNMP] subdirectory of TCPIP$EXAMPLES. Standard .MY files are also provided for your convenience.

The custom MIB definition files have the default extension .MY.

3.3.2. Processing the Input File with the MIB Compiler

Process your ASN.1 source files with the MIB compiler using the DCL command MIBCOMP.

Note

If you are familiar with processing on UNIX systems, you can use the UNIX utilities `snmpi` and `mosy`. See Section 3.3.2.1 for more information.

The compilation process produces two template C programming modules that are used in building the executable subagent code. When you run the compiler, specify all the ASN.1 source files for a given subagent. Whenever any of these source files are updated, you must repeat the compilation process.

The syntax for the MIBCOMP command is as follows:

```
MIBCOMP MIB-source-file "subtree" [/PREFIX=prefix-name] [/PRINT_TREE] [/
SNMPV2]
```

The parameters and qualifiers for the MIBCOMP command are as follows:

Parameter or Qualifier	Definition
MIB-source-file	A comma-separated list of MIB definition files. The standard extension is .MY, but you can specify any valid OpenVMS file name. You must specify the full filename.
subtree	The text name for the root of your MIB definitions. This parameter must be enclosed in quotation marks. This name is used in generating names for template C modules and also for the names of the files themselves: *subtree*_tbl.c and *subtree*_tbl.h.
/PREFIX= *prefix-name*	The MIB compiler attaches the *prefix-name*string to the beginning of all generated names.
/PRINT_TREE	Displays the entire MIB subtree.
/SNMPV2	Specifies the use of SNMP Version 2 parsing rules.

The following is an example of processing the chess example files using the /PRINT_TREE qualifier:

```
$ MIBCOMP RFC1442.MY, CHESS_MIB.MY "chess" /PRINT_TREE
Processing RFC1442.MY
Processing CHESS_MIB.MY
Dump of objects in lexical order
    - This file created by program 'snmpi -p'
ccitt                               0
```

```
iso                                     1
 internet                               1.3.6.1
  directory                             1.3.6.1.1
  mgmt                                  1.3.6.1.2
  experimental                          1.3.6.1.3
  private                               1.3.6.1.4
   enterprises                          1.3.6.1.4.1
    dec                                 1.3.6.1.4.1.36
     ema                                1.3.6.1.4.1.36.2
      sysobjectids                      1.3.6.1.4.1.36.2.15
       decosf                           1.3.6.1.4.1.36.2.15.2
        chess                           1.3.6.1.4.1.36.2.15.2.99
         chessProductID                 1.3.6.1.4.1.36.2.15.2.99.1
ObjectID        read-only
         chessMaxGames                  1.3.6.1.4.1.36.2.15.2.99.2
INTEGER         read-only
         chessNumGames                  1.3.6.1.4.1.36.2.15.2.99.3
INTEGER         read-only
         gameTable                      1.3.6.1.4.1.36.2.15.2.99.4
          gameEntry                     1.3.6.1.4.1.36.2.15.2.99.4.1
indexes: gameIndex
          gameIndex
1.3.6.1.4.1.36.2.15.2.99.4.1.1
INTEGER          read-write
          gameDescr
1.3.6.1.4.1.36.2.15.2.99.4.1.2
DisplayString   read-write              range: 0 to 255
          gameNumMoves
1.3.6.1.4.1.36.2.15.2.99.4.1.3
INTEGER          read-write
          gameStatus
1.3.6.1.4.1.36.2.15.2.99.4.1.4
INTEGER          read-write
                    enum: complete              1
                    enum: underway              2
                    enum: delete                3
         moveTable                      1.3.6.1.4.1.36.2.15.2.99.5
          moveEntry                     1.3.6.1.4.1.36.2.15.2.99.5.1
indexes: gameIndex moveIndex
          moveIndex
1.3.6.1.4.1.36.2.15.2.99.5.1.1
INTEGER          read-write
          moveByWhite
1.3.6.1.4.1.36.2.15.2.99.5.1.2
DisplayString   read-write
                    range: 0 to 255
          moveByBlack
1.3.6.1.4.1.36.2.15.2.99.5.1.3
DisplayString   read-write
                    range: 0 to 255
          moveStatus
1.3.6.1.4.1.36.2.15.2.99.5.1.4
INTEGER          read-write
                    enum: ok                    1
                    enum: delete                2
  security                              1.3.6.1.5
  snmpV2                                1.3.6.1.6
   snmpDomains                          1.3.6.1.6.1
```

```
    snmpProxys                                  1.3.6.1.6.2
    snmpModules                                 1.3.6.1.6.3
joint_iso_ccitt                             2
-------------------------
11 objects written to chess_tbl.c
11 objects written to chess_tbl.h
```

3.3.2.1. UNIX Utilities Supplied with TCP/IP Services

For compatibility with UNIX, the `mosy` and `snmpi` utilities are provided with TCP/IP Services for generating the C language code that defines the object tables. These UNIX utilities are supported on OpenVMS for compatibility with UNIX-developed procedures. For information about using these utilities, refer to the *VSI UNIX Network Programmer's Guide*.

3.3.2.2. Object Tables

The MIBCOMP command is used to generate the C language code that defines the object tables from the MIBs. The object tables are defined in the emitted files *subtree*_TBL.H and *subtree*_TBL.C, which are compiled into your subagent.

These modules are created by the MIBCOMP command or the UNIX utilities. VSI recommends that you do not edit them. If the MIBs change or if a future version of the SNMP development utilities requires your object tables to be rebuilt, it is easier to rebuild and recompile the files if you did not edit them.

3.3.2.3. The *subtree*_TBL.H Output File

The *subtree*_TBL.H file contains the following sections:

1. A declaration of the subtree structure

2. Index definitions for each MIB variable in the subtree

3. Enumeration definitions for MIB variables with enumerated values

4. MIB group data structure definitions

5. Method routine function prototypes

The following sections describe each section of the *subtree*_TBL.H file.

1. Declaration Section

The first section of the *subtree*_TBL.H file is a declaration of the subtree structure. The subtree is automatically initialized by code in the *subtree*_TBL.C file. A pointer to this structure is passed to the `esnmp_register` routine to register a subtree with the master agent. All access to the object table for this subtree is through this pointer. The declaration has the following form:

```
extern SUBTREE subtree_subtree;
```

2. Index Definitions Section

The second section of the *subtree*_TBL.H file contains index definitions for each MIB variable in the *subtree* of the form:

```
#define I_mib-variable nnn
```

These values are unique for each MIB variable in a subtree and are the index into the object table for this MIB variable. These values are also generally used to differentiate between variables that are implemented in the same method routine so they can be used in a switch operation.

3. Enumeration Definitions Section

The third section of the *subtree*_TBL.H file contains enumeration definitions for those integer MIB variables that are defined with enumerated values, as follows:

```
#define D_mib-variable_enumeration-name value
```

These definitions are useful because they describe the value that enumerated integer MIB variables may take on. For example:

```
/* enumerations for gameEntry group */
    #define   D_gameStatus_complete        1
    #define   D_gameStatus_underway        2
    #define   D_gameStatus_delete          3
```

4. MIB Group Data Structure Definitions Section

The fourth section of the *subtree*_TBL.H file contains data structure definitions of the following form:

```
typedef structxxx {
    type          mib-variable;
  .
  .
  .
    char          mib-variable_mark;
  .
  .
  .
    } mib-group_type
```

The MIB compiler generates one of these data structures for each MIB group in the subtree. Each structure definition contains a field representing each MIB variable in the group. In addition to the MIB variable fields, the structure includes a 1-byte *mib-variable-mark* field for each variable. You can use these for maintaining status of a MIB variable. For example, the following is the group structure for the chess MIB:

```
typedef struct _chess_type {
    OID   ches;
    int   chessMaxGames;
    int   chessNumGames;
    char chessProductID_mark;
    char chessMaxGames_mark;
    char chessNumGames_mark;
} chess_type;
```

Although MIB group structures are provided for your use, you are not required to use them. You can use the structure that works best with your method routines.

5. Method Routine Prototypes Section

The fifth section of the *subtree*_TBL.H file describes the method routine prototypes. Each MIB group within the subtree has a method routine prototype defined. A MIB group is a collection of MIB variables that are leaf nodes and that share a common parent node.

There is always a function prototype for the method routine that handles the `Get`, `GetNext`, and `GetBulk` operations. If the group contains any writable variables, there is also a function prototype for the method routine that handles `Set` operations. Pointers to these routines appear in the subtree's object table which is initialized in the *subtree*_TBL.C module. You must write method routines for each prototype that is defined, as follows:

```
extern int mib-group get(METHOD *method );

extern int mib-group set(METHOD *method );
```

For example:

```
extern int chess_get(METHOD *method )

extern int chess_set(METHOD *method );
```

3.3.2.4. The *subtree*_TBL.C Output Files

The *subtree*_TBL.C file contains the following sections:

1. An array of integers representing the OIDs for each MIB variable

2. An array of OBJECT structures

3. An initialized SUBTREE structure

4. Routines for allocating and freeing the `mib_group_type`

The following sections describe each section of the *subtree*_TBL.C file.

1. Array of Integers Section

The first section of the *subtree*_TBL.C file is an array of integers used to represent the OID of each MIB variable in the subtree. For example:

```
static unsigned int elems[] = {
     1, 3, 6, 1, 4, 1, 36, 2, 15, 2, 99,    /* chess */
     1, 3, 6, 1, 4, 1, 36, 2, 15, 2, 99, 1, 0, /* chessProductID */
 . . .
     1, 3, 6, 1, 4, 1, 36, 2, 15, 2, 99, 5, 1, 4, 0, /* moveStatus */
};
```

The first line represents the root of the tree; the other lines represent specific variables. The latter groups are all terminated by a zero, a programming convenience in internal implementations of API routines.

2. Array of OBJECT Structures Section

The second section of the *subtree*_TBL.C file is an array of OBJECT structures. Each MIB variable within the subtree has one OBJECT. The chess example produces the following:

```
static OBJECT objects[] = {
   {I_chessProductID              , {12, &elems[ 11]}, ESNMP_TYPE_ObjectId
   , chess_get, NULL},
 . . .
```

An OBJECT structure represents a MIB variable and has the following fields:

- `object_index` — The constant I_ *mib-variable* from the *subtree*_TBL.H file, which identifies this variable (in the chess example, `I_chessProductID`.)

- `oid` — The variable's OID (points to a part of `elems[]`).

  This variable is of type OID, which is a structure containing two elements: the number of elements in the OID and a pointer to the correct starting place in the array of elements (`elems[]` in the chess example).

  In the chess example, `oid` is designated by `{12, &elemens[ 11]}`. This indicates that:

  - The OID has 12 integers separated by dots in the ASCII text representation (`"1.3.6.1.4.1.36.2.15.2.99.2"`)

  - The integer with index 11 in the array `elems[]` is the first element.

- `type` — The variable's eSNMP data type.

- `getfunc` — The address of the method routine to call for `Get` requests (null if no routine exists).

- `setfunc` — The address of the method routine to call for `Set` requests (null if no routine exists).

The master agent does not access object tables or MIB variables directly. It only maintains a registry of subtrees. When a request for a particular MIB variable arrives, it is processed as shown in the following steps (where the MIB variable is `mib_var` and the subtree is `subtree_1`):

1. The master agent finds `subtree_1` as the authoritative region for the `mib_var` in the register of subtrees. The authoritative region is determined as the registered MIB subtree that has the longest prefix and the highest priority.

2. The master agent sends a message to the subagent that registered `subtree_1`.

3. The subagent consults its list of registered subtrees and locates `subtree_1`.

   It searches the object table of `subtree_1` and locates the following:

   - `mib_var` (for `Get` and `Set` routines)

   - The first object lexicographically after `mib_var` (for `Next` or `Bulk` routines)

4. The appropriate method routine is called. If the method routine completes successfully, the data is returned to the master agent. If the method routine fails when doing a `Get` or `Set`, an error is returned. If the method routine fails when doing a `GetNext`, the code keeps trying subsequent objects in the object table of `subtree_1` until either a method routine returns successfully or the table is exhausted. In either case, a response is returned.

5. If the master agent detects that `subtree_1` could not return data on a `Next` routine, it recursively tries the subtree lexicographically after `subtree_1` until a subagent returns a value or the registry of subtrees is exhausted.

3. Initialized Subtree Structure Section

The third section of the *subtree*_TBL.C file is the SUBTREE structure itself. A pointer to this structure is passed to the eSNMP library routine `esnmp_register` to register the subtree. It is

through this pointer that the library routines find the object structures. The following is an example of the chess subtree structure:

```
SUBTREE chess_subtree = { "chess", "1.3.6.1.4.1.36.2.15.2.99",
                          { 11, &elems[0] }, objects, I_moveStatus};
```

The following table describes the elements of the SUBTREE structure, the definition of each element in the header file (*subtree*_TBL.H)), and the element in the chess example:

Description	Header File Representation	Example
The name of the subtree's base element.	name	"chess"
The ASCII string representation of the subtree's OID. This is what actually gets registered.	dots	"1.3.6.1.4.1.36.2.15.2.99"
The OID structure for the base node of the subtree. This points back to the array of integers.	oid	11, &elems[0] }
A pointer to the array of objects in the object table. It is indexed by the I_*xxxx*definitions found in the *subtree*_TBL.H file.	object_oid	objects
The index of the last object in the *object*_TBL file. This is used to determine when the end of the table has been reached.	last	I_moveStatus

4. Routines Section

The final section of the *subtree*_TBL.C file. Contains short routines for allocating and freeing the `mib_group_type`. These are provided as a convenience and are not a required part of the API.

3.4. Including the Routines and Building the Subagent

The MIB compiler does not generate code for implementing the method routines for your subagent. This includes code for processing `get`, `set`, and other SNMP requests as well as for generating traps. You must write this code yourself. See the CHESS_MIB.C module for an example.

To produce executable subagent code, follow these steps:

1. Compile the C modules generated by the MIB compiler, along with your implementation code. Use a command in the following format (derived from the sample provided for building the chess example in TCPIP$BUILD_CHESS.COM):

    ```
    $ CC /INCLUDE=TCPIP$SNMP /PREFIX=ALL /STANDARD=VAX CHESS_METHOD.C, -
    _$ CHESS_MIB.C, CHESS_TBL.C
    ```

    Depending on the version of the C compiler you are using, you might see warnings that you can ignore. Portions of these warnings are as follows:

    ```
    %CC-I-SIGNEDKNOWN   In this declaration, DEC C recognizes the ANSI
    ```

```
                        keyword "signed". This differs from the VAX C
  behavior.

%CC-I-INTRINSICINT  In this statement, the return type for intrinsic
                    "strlen" is being changed from "size_t" to "int".
```

2. Link the object modules using a command and options in the following format (derived from the chess example):

```
$ LINK SYS$INPUT/OPTIONS
CHESS_METHOD.OBJ
CHESS_MIB.OBJ
CHESS_TBL.OBJ
SYS$SHARE:TCPIP$ESNMP_SHR.EXE/SHARE
```

To link with the eSNMP object library, enter the following command:

```
$ LINK SYS$INPUT/OPTIONS
CHESS_METHOD.OBJ
CHESS_MIB.OBJ
CHESS_TBL.OBJ
TCPIP$SNMP:TCPIP$ESNMP.OLB/LIBRARY
TCPIP$LIBRARY:TCPIP$LIB.OLB/LIBRARY
```

Alternatively, you can link your subagent with the eSNMP API shareable image (TCPIP $ESNMP_SHR.EXE). The resulting executable image is smaller and can be run without relinking against any future versions of the shareable image. To link the example object with the shareable image, enter the following command:

```
$ LINK SYS$INPUT/OPTIONS
CHESS_METHOD.OBJ
CHESS_MIB.OBJ
CHESS_TBL.OBJ
SYS$SHARE:TCPIP$ESNMP_SHR.EXE/SHARE
```

3.5. Including Extension Subagents in the Startup and Shutdown Procedures

You can add your custom subagents to the SNMP startup and shutdown procedures by editing the following files:

File Name	Edit Required
TCPIP$EXTENSION_MIB_STARTUP.COM	Edit the example lines to include an INSTALL CREATE command for custom images that need to be installed, possibly with privileges. Remove extra example lines, and adjust the GOTO statement.
TCPIP$EXTENSION_MIB_RUN.COM	Edit the example lines to include a RUN command for custom images. Remove extra example lines, and adjust the GOTO statement.
TCPIP$EXTENSION_MIB_SHUTDOWN.COM	Edit the example lines to: • Include symbols for the detached processes that are running custom images. Use the

File Name	Edit Required
	same process names specified in TCPIP$EXTENSION_MIB_RUN.COM. • Modify the IF and THEN statements to include the new symbols. • Include an INSTALL DELETE command for images installed in TCPIP$EXTENSION_MIB_STARTUP.COM. • Remove extra example lines, and adjust the GOTO statement.

Chapter 4. Using the SNMP Utilities

TCP/IP Services includes the following programs, which are useful for testing applications and for analyzing SNMP problems:

- TCPIP$SNMP_REQUEST (MIB browser) (Section 4.1)

- TCPIP$SNMP_TRPSND (trap sender) (Section 4.2)

- TCPIP$SNMP_TRPRCV (trap receiver) (Section 4.2)

These programs can be invoked by commands that are defined by the SYS$STARTUP:TCPIP$DEFINE_COMMANDS.COM command procedure.

This chapter describes how to use the supplied SNMP utilities.

4.1. Using the MIB Browser

TCP/IP Services provides the `snmp_request` MIB browser that acts as a simple client to handle single SNMP requests for reading and writing to a MIB. The browser sends SNMP Version 1 and SNMP Version 2 request PDUs to an agent and displays the agent's response.

To run the MIB browser, follow these steps:

1. Define a foreign command for the program:

   ```
   $ snmp_request == "$SYS$SYSTEM:TCPIP$SNMP_REQUEST"
   ```

   Alternatively, you can run the SYS$MANAGER:TCPIP$DEFINE_COMMANDS.COM procedure to define all the foreign commands available with TCP/IP Services.

2. Enter the command using the following format.

   ```
   snmp_request agent "community" request_type [flags] variable [data-type
    value]
   ```

   Section 4.1.1 describes the parameters. Section 4.1.2 describes the flags.

4.1.1. MIB Browser Parameters

Table 4.1 describes the `snmp_request` parameters.

Table 4.1. snmp_request Command Parameters

Parameter	Function
agent	The host name or IP address (in dot notation) of the managed node to query.
	If you specify 0, 0.0.0.0., 127.0.0.1, or "localhost,"the server on the browser's host is queried.
"community"	The community string to be used in the query. This parameter is case sensitive. Typically, agents are configured to permit read access to the community string "public". For accurate interpretation, be sure to enclose the name in quotation marks (" ").

Parameter	Function	
	Note that if you do not use quotation marks, the name is changed to lowercase.	
`request-type`	PDU type to send. Can be one of the following SNMP requests:	
	`Get`	Sends a Get-Request PDU.
	`GetNext`	Sends a GetNext-Request PDU.
	`GetBulk`	Sends a GetBulk-Request PDU (SNMP Version 2 only).
	`Set`	Sends a Set-Request PDU.

`variable`	An object identifier (OID) in ASN.1 notation that is associated with an object in a MIB. For example:
	
	`$ snmp_request host1 "public" getnext -d`
	`  1.3.6.1.6.3.1.1.6`	

`data-type`	Data type of the *value*. This parameter can be specified for `Set` requests. The data types are described in Section 4.1.3.	
`value`	The value to which to set the contents of the OID. This parameter is used for `set` requests.	

For `Set` requests, you can specify more than one group of the following:

- `variable-name`

- `data-type`

- `value`

For other requests, you can specify more than one variable name, except when you specify the `-l` or `-t` flag; these flags are valid only with a `GetNext` or `GetBulk` request, for which only one OID is permitted.

4.1.2. MIB Browser Flags

Flags are specified in UNIX format.

Because flags and data types are case sensitive, you should always enter them in the case that is specified. If a letter or value is specified as uppercase, you must enclose it in quotation marks. In general, if you use uppercase letters where lowercase is specified, the results are unpredictable. For example, the flag `"-v2C"` functions correctly but the flag `"-V2c"` does not, because the flag character (v) must be lowercase.

If you do not specify a flag, or if you specify an invalid flag, a usage message is displayed. You must place the flags after the `request-type` parameter.

Table 4.2 describes the flags for the `snmp_request` command.

Table 4.2. Flags for the snmp_request Command

Flag	Description
`-d`	Specifies hexadecimal dump mode. Before displaying a return value, displays a hexadecimal dump of SNMP packets sent and received. For example:

Flag	Description
| | ```
$ snmp_request host1 "public" getnext -d -v 2c
 1.3.6.1.6.3.1.1.6

Sent:

30290201 01040670 75626C69 63A11C02
 0).....public...
047BE9C1 BD020100 02010030 0E300C06   .
{.........0.0..
082B0601 06030101 060500              .+.........

Received:

30820033 02010104 06707562 6C6963A2
 0..3.....public.
2602047B E9C1BD02 01000201 00308200   &..
{.........0..
16308200 12060A2B 06010603 01010601   .0.....
+........
00020478 D917FC                       ...x...
1.3.6.1.6.3.1.1.6.1.0 = 2027493372
``` |
| `-i max_ignores` | Specifies the number of times the MIB browser listens for a reply packet to a request if it receives an invalid packet (caused by an invalid packet identifier, version, or SNMP version and command combination). Specify a positive integer for the value (*max_ignores*). If you specify a negative value, it will be converted to an unsigned positive integer. If you specify 0, no retries are attempted.

If, after an invalid reply packet is received, a valid reply packet is received, the ignore counter is reset to the value of *max_ignores*.

If a timeout occurs after an invalid packet is received, the packet is resent, the resend counter is decremented, and the ignore counter is reset to the value of *max_ignores*.

You cannot use the `-i` flag when you perform a query with the `-l` or `-t` flags to automatically increment the input OID and continue querying a server after a general SNMP error has occurred, as may happen with a faulty server. In this case, the query is terminated even though the end of the MIB selection has not been reached. You must manually increment the input OID to skip the error and continue with the query. |
| `-l` | Specifies loop mode. Note that this flag is the letter l, not the number 1.

Valid only if `request-type` is `GetNext` or `GetBulk` (where flag n is set to 0, and flag m is set to a number greater than 0).

Causes the master agent to traverse all the MIBs registered with the master agent, starting at the first OID after the one specified in the command. (Note that you can specify only one `variable-name`[OID].) Responses are received one at a time, and for each |

Flag	Description

	one, the OID returned by the master agent is used in a subsequent request. Corresponds to the behavior of standard `mibwalk` programs.
	
	The MIB browser reads and displays responses, and issues requests until the master agent has no more data, times out, or you press Ctrl/Y or Ctrl/C.
	
	If specified with the `GetBulk` request, the `-n` and `-m` flags and associated values are ignored, and the behavior is identical to that of `GetNext`.
	
	When the last OID handled by the master agent is reached, you receive a response similar to the following for a query on OID 1.3.6.1.6.3.1.1.6.1 using SNMP Version 1:
	
	`1.3.6.1.6.3.1.1.6.1.0 = 693056825`
	`- no such name - returned for variable 1`
	
	For a query using SNMP Version 2, the example response is:
	
	`1.3.6.1.6.3.1.1.6.1.0 = 693056825`
	`1.3.6.1.6.3.1.1.6.1.0 = - end of mib view -`
	
	These examples assume that:
	
	• OID 1.3.6.1.6.3.1.1.6.1.0 is the last OID supported on the target host.
	
	• The target host is running an SNMP Version 2 agent.
	
	The statement `end of mib view` refers to OIDs for all MIBs registered with the master agent.

`-m max_repetitions`	Specifies the number of repetitions requested for repeating variables. Applies only to the `GetBulk` and `GetNext` requests. Note that the resulting display can be confusing because the results for the repeater OIDs are interleaved. That is, the OIDs are displayed in alternate progression for faster memory throughput. If you specify `GetBulk` without specifying both the `-m` and `-n` flags, the results are unpredictable.
`-n non_repeaters`	Specifies the number of variables for which no repetition is requested. Applies only to the `GetBulk` request. If you specify `GetBulk` without specifying both the `-m` and `-n` flags, the results are unpredictable.
`-p port`	Specifies the port where the request is to be sent. If not specified, the request is sent to well-known SNMP port 161.
`-r max_retries`	Specifies the number of times the MIB browser resends a request packet if it times out before receiving a reply. Specify a positive integer for the value (*max_retries*). If you specify a negative value, it will be converted to an unsigned positive integer. If you specify 0, no retries are tried.

Flag	Description
	If, after a timeout and a resend, a reply packet is received, the resend counter is reset. After another timeout, the specified number of *max_retries* is sent.
-s *sleep_interval*	Specifies the number of seconds between iterations of sending a request (for the -r flag) and listening for a reply (for the -i flag). The default is 1 second. This flag is ignored if neither the -r flag nor the -i flag are specified. The -s flag is useful for specifying a time to wait between resends, which might be necessary when a server agent is starting up.
-t	Specifies tree mode. Valid only if *request-type* is GetNext or GetBulk (where flag n is set to 0 and flag m is set to a number greater than 0). Similar to the -l flag. Directs the agent to perform a MIB walk for the subtree with the *variable_name* as its root. The program reads and prints responses and issues requests until the agent has no more data for the specified subtree, times out, or is interrupted by a user.
-v *version*	Specifies the SNMP version to use for sending the PDU. The versions are: 2c or 1 (default). Not case sensitive. You can specify the flag without a space (-v2c and -v1). If *request_type* is getbulk, the version defaults to SNMP Version 2. If you specify -v 2c to send a message to an SNMP Version 1 agent or subagent, it is unlikely to respond.
-w *max_wait*	Specifies the maximum seconds the snmp_request program waits for a reply before timing out. Cannot be 0. The default is 3.

The -i, -r, and -s flags apply to individual queries. If you specify the -l or -t flags also, the values for the -i, -r, and -s flags are applied to each iteration.

4.1.3. MIB Browser Data Types

The snmp_request and snmp_trapsnd commands support the data types listed in Table 4.3. These values apply to Set requests only.

Table 4.3. Data Types for the snmp_request and snmp_trapsnd Commands

Data Type	Value
Counter	-c
Counter64 [1]	-l
Display string	-D
Gauge	-g
Integer	-i
IP address	-a
NULL	-N

Data Type	Value
Object identifier	`-d`
Octet	`-o`
Opaque string	`-q`
Time ticks	`-t`

[1]For support of trap sender program (TCPIP$SNMP_TRAPSND.EXE) only. Properly defined, MIB variables of type Counter64 are not writable.

Note

Except for `-l` (Counter64), the data types are case sensitive. To preserve uppercase for display strings and NULL, enclose the value in double quotation marks. For example, "--D" or "--N".

On OpenVMS Alpha systems, you must specify the value of the `-l` data type as a 64-bit integer. For example:

```
$ snmp_trapsnd 0.0 mynode 6 33 100 -h mynode -v2c -
_$ 1.3.6.1.2.1.1.4.0 "l" 1311768467294899695
```

On OpenVMS VAX systems, you must specify the value of the `-l` data type as a 16-digit hexadecimal value. For example:

```
$ snmp_trapsnd 0.0 mynode 6 33 100 -h mynode -v2c -
_$ 1.3.6.1.2.1.1.4.0 "l" 0x1234567890ABCDEF
```

Note that alphabetic characters are not case sensitive when used with the `-l` data type.

For more information about the data types, see RFCs 1155 and 1902.

4.1.4. Command Examples for snmp_request

This section presents several examples of using the `snmp_request` utility. In the following `snmp_request` command examples:

- The valid host name is `marley.dec.com`.

- The `"public"` community is type Read, address 0.0.0.0.

- The `"address_list"` community is type Read and Write, with write access for the host on which the `snmp_request` program is running.

- The `location` has been specified as shown in the following command:

  ```
  TCPIP> SET CONFIGURATION SNMP -
  _TCPIP> /LOCATION=(FIRST="Falcon Building", SECOND="Los Angeles, CA")
  ```

- The command responses have been edited for readability.

Examples

1. The following example shows how to retrieve the value of the MIB II variable `sysDescr.0` (`1.3.6.1.2.1.1.1.0`). The request is successful because the OID (`variable_name`) provided in the command line exists and is readable. This OID is returned by the subagent code that resides in the master agent.

   ```
   $ snmp_request marley.dec.com "public" get 1.3.6.1.2.1.1.1.0
   ```

```
1.3.6.1.2.1.1.1.0 = marley.dec.com AlphaServer 2100 4/200 OpenVMS
    V7.1 Digital TCP/IP Services for OpenVMS
```

2. The following example shows how to retrieve two MIB II variables. This example is identical to the previous example, except that two OID values are input and two returned: instance 1 of `ifDescr` and `hrSystemUptime`. Note that the first value comes from the MIB II subagent (TCPIP$OS_MIBS) and the second comes from the Host Resources MIB subagent (TCPIP $HR_MIB).

```
$ snmp_request marley.dec.com "public" get 1.3.6.1.2.1.2.2.1.2.1 -
_$ 1.3.6.1.2.1.25.1.1.0

1.3.6.1.2.1.2.2.1.2.1 = LO IP Interface: LO0, OpenVMS Adapter: <none>,
    Loopback Port
1.3.6.1.2.1.25.1.1.0 = 6024942 = 0 d 16:44:9
```

3. The following example shows how to retrieve the next MIB II variable. This is similar to the command in example 1, except that a `GetNext` request is issued and `sysObjectID.0` (1.3.6.1.2.1.1.2.0) is returned.

```
$ snmp_request marley.dec.com "public" getnext 1.3.6.1.2.1.1.1.0

1.3.6.1.2.1.1.2.0 = 1.3.6.1.4.1.36.2.15.13.7.1
```

4. The following example shows how to use the SNMP Version 2 `GetBulk` request to retrieve the MIB II variables `sysUpTime.0` (1.3.6.1.2.1.1.1.0) and `sysDescr.0` (1.3.6.1.2.1.1.2.0), and for the first three interfaces, the values of `ifDescr` (OIDs with the prefix 1.3.6.1.2.1.2.2.1.2) and `ifType` (OIDs with the prefix 1.3.6.1.2.1.2.2.1.3).

```
$ snmp_request marley.dec.com "public" getbulk -n 2 -m 3 -

_$ 1.3.6.1.2.1.1.1 1.3.6.1.2.1.1.2 -
_$ 1.3.6.1.2.1.2.2.1.1 1.3.6.1.2.1.2.2.1.2 1.3.6.1.2.1.2.2.1.3

Warning: using version SNMPv2 for getbulk command.
1.3.6.1.2.1.1.1.0 = marley.dec.com AlphaStation 255/300
    OpenVMS V7.1 Digital TCP/IP Services for OpenVMS
1.3.6.1.2.1.1.2.0 = 1.3.6.1.4.1.36.2.15.13.7.1
1.3.6.1.2.1.2.2.1.1.1 = 1
1.3.6.1.2.1.2.2.1.2.1 = LO IP Interface: LO0, OpenVMS Adapter: <none>,
    Loopback Port1.3.6.1.2.1.2.2.1.3.1 = 24
1.3.6.1.2.1.2.2.1.1.3 = 3
1.3.6.1.2.1.2.2.1.2.3 = WE IP Interface: WE0, OpenVMS Adapter: EWA0:,
    PCI bus Ethernet Adapter
1.3.6.1.2.1.2.2.1.3.3 = 6
1.3.6.1.2.1.2.2.1.1.4 = 4
1.3.6.1.2.1.2.2.1.2.4 = WF IP Interface: WF0, OpenVMS Adapter: FWA0:,
    DEFPA PCI bus FDDI Adapter
1.3.6.1.2.1.2.2.1.3.4 = 15
```

5. The following example shows how to use the `GetNext` request with the `-l` (loop) flag to retrieve all OIDs starting at the first instance after the OID input and finishing at the end of the MIB view. Note that if an SNMP Version 2 agent is the server, the results using `getbulk` are identical (in general, SNMP Version 1 agents do not support `getbulk` requests).

```
$ snmp_request marley.dec.com "public" getnext -l 1.3.6.1.2.1.1.1.0
1.3.6.1.2.1.1.2.0 = 1.3.6.1.4.1.36.2.15.13.7.1
```

```
1.3.6.1.2.1.1.3.0 = 1280260 = 0 d 3:33:22
1.3.6.1.2.1.1.4.0 = Sam Spade
1.3.6.1.2.1.1.5.0 = marley.dec.com
1.3.6.1.2.1.1.6.0 = Falcon Building Los Angeles, CA
1.3.6.1.2.1.1.7.0 = 721.3.6.1.2.1.1.8.0 = 0 = 0 d 0:0:0
.
.
.
1.3.6.1.2.1.25.5.1.1.2.294 = 560
1.3.6.1.2.1.25.5.1.1.2.295 = 472
1.3.6.1.6.3.1.1.6.1.0 = 1296505215
- no such name - returned for variable 1
```

6. The following example uses the same command as in example 5, but it specifies the `-t` flag instead of the `-l` flag. Only OIDs with the prefix matching the input OID are returned. Note that as with other `getnext` request examples, the value for the input OID is not returned. If an SNMP Version 2 agent is the server, the results using `getbulk` are identical.

```
$ snmp_request marley.dec.com "public" getnext -t 1.3.6.1.2.1.1

1.3.6.1.2.1.1.2.0 = 1.3.6.1.4.1.36.2.15.13.7.1
1.3.6.1.2.1.1.3.0 = 1302232 = 0 d 3:37:2
1.3.6.1.2.1.1.4.0 = Sam Spade1.3.6.1.2.1.1.5.0 = marley.dec.com
1.3.6.1.2.1.1.6.0 = Falcon Building Los Angeles, CA
1.3.6.1.2.1.1.7.0 = 721.3.6.1.2.1.1.8.0 = 0 = 0 d 0:0:0
1.3.6.1.2.1.1.9.1.2.1 = 1.3.6.1.4.1.36.15.3.3.1.1
1.3.6.1.2.1.1.9.1.2.2 = 1.3.6.1.4.1.36.15.3.3.1.2
1.3.6.1.2.1.1.9.1.3.1 = Base o/s agent (OS_MIBS) capabilities
1.3.6.1.2.1.1.9.1.3.2 = Base o/s agent (HR_MIB) capabilities
1.3.6.1.2.1.1.9.1.4.1 = 0 = 0 d 0:0:0
1.3.6.1.2.1.1.9.1.4.2 = 0 = 0 d 0:0:0
```

7. The following example shows how to send a `Set` request. The request succeeds because the command line specifies the correct type for the variable, and all the conditions for enabling `Set` requests are met on the server.

```
$ snmp_request marley.dec.com "address_list" -
_$ set 1.3.6.1.2.1.1.4.0 "D" "Richard Blaine"
1.3.6.1.2.1.1.4.0 = Richard Blaine
```

8. The following example shows how to display the contents of packets that are sent and received. Note that only the SNMP-specific portion of the UDP packets is displayed.

```
$ snmp_request marley.dec.com "public" get -d 1.3.6.1.2.1.1.4.0

Sent:

 3082002D 02010004 06707562 6C6963A0    0..-.....public.
 2002047B E9C1BD02 01000201 00308200     ..{..........0..
 10308200 0C06082B 06010201 01040005    .0.....+........
 00                                      .

Received:

 3082003B 02010004 06707562 6C6963A2    0..;.....public.
 2E02047B E9C1BD02 01000201 00308200    ...{..........0..
 1E308200 1A06082B 06010201 01040004    .0.....+........
 0E526963 68617264 20426C61 696E65      .Richard Blaine
```

```
1.3.6.1.2.1.1.4.0 = Richard Blaine
```

4.2. Using the Trap Sender and Trap Receiver Programs

TCP/IP Services provides the following programs that allow you to set up a simple client on your system to send and receive trap messages:

- `snmp_trapsnd` (TCPIP$SNMP_TRAPSND.EXE)

  Sends SNMP Version 1 and SNMP Version 2 trap messages. Use only for testing or to send significant state changes that occur on the managed node.

- `snmp_traprcv` (TCPIP$SNMP_TRAPRCV.EXE)

  Listens for SNMP trap messages and displays any it receives.

By default, these programs use UDP port 162. However, you can specify another port with the `-p` flag or set up an SNMP-trap service that specifies the port you want to use. Note, however, that the use of UDP port 162 is coded into standard MIBs.

Both programs support the use of the UDP (default) and TCP transports. However, the standard TCP/IP subagents and the Chess example use UDP only. Therefore, if you specify the `-tcp` flag when you enter the `snmp_traprcv` command, the program uses TCP to process traps only from the trap sender program or from a user application written to use TCP.

The following sections explain how to enter commands for both programs. Because flags and data types are case sensitive, you should always enter them in the case that is specified. If a letter or value is specified as uppercase, you *must* enclose it in quotation marks. In general, if you use uppercase letters where lowercase is specified, the results are unpredictable. For example, flag `"-v2C"` functions correctly but flag `"-V2c"` does not, because the flag character (`v`) must be lowercase.

The trap receiver does not use the trap communities specified using the TCPIP$CONFIG.COM command procedure or any configuration file. You set the trap communities using the trap sender program. Use the `c` flag to specify the community name, and the `-h` flag to set the host name. For more information about using these flags, see Section 4.2.1.2.

4.2.1. Entering Commands for the Trap Sender Program

The trap sender program lets you send SNMP Version 1and SNMP Version 2 trap messages. You should use this program only when you want to test the client or when significant state changes occur on the managed node.

The trap sender program encodes an SNMP Version 1 trap PDU (see RFCs 1155, 1156, 1157, and 1215) or an SNMP Version 2 trap PDU (see RFCs 1905 and 1908) into an SNMP message and sends it to the specified hosts. You use parameters and flags to specify the data fields in the trap PDU.

Traps are uniquely identified in the PDU, as follows:

- SNMP Version 1 is identified by a combination of parameters.

- SNMP Version 2 is identified by the value of `snmpTrapOID`.

To run the trap sender program, do the following:

1. Define a foreign command for the program:

   ```
   $ snmp_trapsnd == "$SYS$SYSTEM:TCPIP$SNMP_TRAPSND"
   ```

   Alternatively, you can run the SYS$MANAGER:TCPIP$DEFINE_COMMANDS.COM
   procedure to define all the foreign commands available with TCP/IP Services.

2. Enter a command using the following format:

   ```
   snmp_trapsnd enterprise agent generic-trap specific-trap timeticks
   [-v version] [-c community] [-d] [-h host] [-p port] [-tcp]
   [variable_name [data-type value]]
   ```

4.2.1.1. Trap Sender Parameters

Table 4.4 describes the `snmp_trapsnd` parameters. Each parameter is required, but you can specify zero, as appropriate.

Table 4.4. Parameters for the snmp_trapsnd Command

Parameter	Description
enterprise	For SNMP Version 1, specifies the enterprise object identifier (OID) on whose behalf the trap is being sent. For example, 1.3.6.1.4.1.1. If you specify 0 or 0.0, the null OID (0.0) is sent. Make sure that any OID you specify conforms to the OID rules. For SNMP Version 2, when specified with the `-v2c` flag, represents the value of `snmpTrapOID.0`.

agent	For SNMP Version 1 traps. Specifies the host name or IP address of the entity on whose behalf the trap is being generated.
	
	The value for the `agent` field is that of the primary interface for the host on which the master agent (TCPIP$ESNMP_SERVER) is running. You can obtain this address using the following DCL command:
	
	`$ SHOW LOGICAL TCPIP$INET_HOSTADDR`
	
	You can specify the name `local`, which is the same as specifying 0, 0.0, 0.0.0, or 0.0.0.0. In these cases, the address 0.0.0.0 is sent as the agent address in the SNMP Version 1 trap PDU.
	
	To obtain the value of the local host, enter the following TCP/IP management command:
	
	`TCPIP> SHOW CONFIGURATION COMMUNICATION`
	
	If the information is not in address format, enter the following command:
	
	`TCPIP> SHOW HOST/LOCAL`
	
	If the `-v2c` flag is specified, this parameter is ignored.

Parameter	Description
generic-trap	For SNMP Version 1, specifies the generic trap identifier in the form of a number. Must be one of the following values:

SNMP Version 1 Value	Object
0	`coldStart`
1	`warmStart`
c	`linkDown`
3	`linkUp`
4	`authenticationFailure`
5	`egpNeighborLoss`
6	`enterpriseSpecific`

	For SNMP Version 2, when the `-v2c` flag is specified, this parameter must contain a valid OID or 0 as the value of `snmpTrapOID`.
specific-trap	For SNMP Version 1, specifies the enterprise-specific trap number. A numeric value greater than 0 must be present but is ignored if the `-v2c` flag is present or if *generic-trap* is a value other than 6 (`enterpriseSpecific`).
timeticks	Specifies the timestamp value associated with the generation of the trap message. The timestamp value is the current time in units of TIMETICKS (1/100 of a second) since the sending SNMP entity started. A value of 0 causes `snmp_trapsnd` to send the time in hundredths of a second since the operating system was last booted.
variable_name | *data-type value*	Specifies a list of MIB variables to be included in the trap message. For a list of supported values, including a value for the `Counter64` data type, see Table 4.3.

4.2.1.2. Trap Sender Flags

Table 4.5 describes the `snmp_trapsnd` flags.

Table 4.5. Flags for the snmp_trapsnd Command

Flag	Description
`-c` *community*	Specifies a community string to be used when sending the trap. The default is public.
`-d`	Displays a hexadecimal dump of the encoded packet.
`-h` *host*	Specifies the host name or IP address (in ASN.1 dot notation format) of the destination host to receive the trap message. The default is `localhost` (127.0.0.1).
`-p` *port*	Specifies a port number on the destination host where the message is to be sent. The default is UDP 162.
`-tcp`	Specifies that the TCP transport be used instead of the default UDP transport. If a connection

Flag	Description
	cannot be established, the program displays the warning `connect - : connection refused`.
`-v version`	Specifies the SNMP version to use for sending the PDU. The valid versions are `2c` or `1` (default). You can specify the flag and value without including a space (for example, `-v2c` and `-v1`).

4.2.1.3. Trap Sender Examples

In the following `snmp_trapsnd` command examples:

- The first line is the `snmp_trapsnd` command.

- The remainder is the display received when running the trap receiver program (`snmp_traprcv`) without flags included.

1. The following example generates a trap that originated on the `localhost` (specified by the `agent` parameter) using the default SNMP version (SNMP Version 1). The `-h host` parameter is not specified, so the trap will be sent to the local host.

```
$ snmp_trapsnd 0.0 local 0 0 0

Message received from 127.0.0.1

SNMPv1-Trap-PDU:

community -  7075626C 6963                              public
enterprise - 0.0
agent address - 0.0.0.0
trap type - Cold Start (0)
timeticks - 51938978
```

2. The following example generates the same trap as in example 1, except that it specifies the use of SNMP Version 2.

```
$ snmp_trapsnd 0.0 local  0 0 0 "-v2c"
Message received from 127.0.0.1
SNMPv2-Trap-PDU:community -  7075626C 6963
 public

sysUpTime.0 - 51938968 = 6 d 0:16:29
snmpTrapOID.0 - 0.0
```

3. The following example sends values to the node `mynode` with the community name `special`.

```
$ snmp_trapsnd 1.2.3 marley.dec.com 6 33 100 -c special -h mynode

Message received from 16.20.208.68

SNMPv1-Trap-PDU:
community -  73706563 69616c              special

enterprise - 1.2.3
```

```
agent address - 6.20.208.53
trap type - Enterprise-specific (6)
enterprise-specific value - (33)
timeticks - 100
```

4.2.2. Entering Commands for the Trap Receiver Program

The trap receiver program lets you listen for, receive, and display SNMP trap messages. Until interrupted, the program continues to listen on the specified port.

If you enter commands using the default port number or another privileged port number, you must run the program from a privileged account.

To run the trap receiver program, do the following:

1. Define a foreign command for the program:

   ```
   $ snmp_traprcv == "$SYS$SYSTEM:TCPIP$SNMP_TRAPRCV"
   ```

   Alternatively, you can run SYS$MANAGER:TCPIP$DEFINE_COMMANDS.COM to define all the foreign commands available with TCP/IP Services.

2. Enter a command using the following format:

   ```
   snmp_traprcv [-d] [-tcp] [-p port]
   ```

4.2.2.1. Trap Receiver Flags

Table 4.6 describes the `snmp_traprcv` flags.

Table 4.6. snmp_traprcv Command Flags

Flag	Description
-d	Displays a hexadecimal and formatted dump of the received packet.
-p port	Specifies the port number on the local host on which to listen for trap messages. The default is 162.
-tcp	Listens on the TCP port instead of the UDP (default) port. Reads only a single PDU on an established connection, which is similar to the behavior using UDP.

4.2.2.2. Setting Up an SNMP Trap Service

To set up an SNMP trap service for use with the trap receiver program, enter a management command in the following format:

```
SET SERVICE SNMP-TRAP /PROTOCOL=UDP /USER_NAME=TCPIP$SNMP
/PROCESS_NAME=TCPIP$SNMP-TRAP /FILE=TCPIP$SYSTEM:TCPIP$SNMP-TRAP.COM
```

In this command, port 170 is used as an alternative for port 162, and traps that are received on port 162 are ignored.

If you omit the /PROTOCOL qualifier or you use /PROTOCOL=TCP, the service uses the TCP transport. In this case, when you enter a command to run the trap receiver program, you must include the `-tcp` flag.

With the SNMP trap service in place, the trap receiver program queries the service for the port number instead of using the default port 162. If you specify a privileged port number (less than 1024) with the /PORT qualifier, make sure you install the trap receiver program with privileges, or run the program from an account that has SYSPRV privilege. Note that the port number must be greater than zero.

4.2.2.3. Trap Receiver Examples

1. The following example requests trap information on a system that does not have traps configured and does not have SYSPRV privilege or sufficient privilege.

```
$ snmp_traprcv
No snmp-trap service entry, using default port 162.
bind - : permission denied
```

2. The example, supplied from a non-privileged account, requests trap information in hexadecimal dump format on port 1026.

```
$ snmp_traprcv -d -p 1026

Message received from 127.0.0.1

3082002A 02010004 06707562 6C6963A4    0..*.....public.
1D060547 81AD4D01 40040000 00000201    ...G..M.@.......
00020100 4304032D AED23082 0000        ....C..-..0...
SNMPv1-Trap-PDU:

community -  7075626C 6963                              public

enterprise - 0.0
agent address - 0.0.0.0
trap type - Cold Start (0)
timeticks - 53325522
```

Chapter 5. eSNMP API Routines

This chapter provides reference information about the following types of application programming interface (API) routines in the eSNMP developer's kit:

- Interface routines (Section 5.1)

- Method routines (Section 5.2)

- Support routines (Section 5.6)

5.1. Interface Routines

The interface routines are for developers writing the portion of the application programming interface (API) that handles the connection between the agent and the subagent. The interface routines are listed Table 5.1 and described in the following pages.

Table 5.1. Interface Routines

Routine	Function
`esnmp_init`	Initializes the subagent and initiates communication with the master agent.
`esnmp_register`	Requests local registration of a MIB subtree.
`esnmp_unregister`	Cancels local registration of a MIB subtree.
`esnmp_register2`	Requests cluster registration of a MIB subtree.
`esnmp_unregister2`	Cancels cluster registration of a MIB subtree.
`esnmp_capabilities`	Adds a subagent's capabilities to the master agent's `sysORTable`.
`esnmp_uncapabilities`	Removes a subagent's capabilities from the master agent's `sysORTable`.
`esnmp_poll`	Processes a pending request from the master agent.
`esnmp_are_you_there`	Requests a report from the master agent to indicate it is up and running.
`esnmp_trap`	Sends a trap message to the master agent.
`esnmp_term`	Sends a close message to the master agent.
`esnmp_sysuptime`	Converts UNIX system time into a value with the same time base as `sysUpTime`.

esnmp_init

esnmp_init — Initializes the Extensible SNMP (eSNMP) subagent and initiates communication with the master agent.

Syntax

```
int esnmp_init (int *socket, char *subagent_identifier ) ;
```

Arguments

socket

The address of the integer that receives the socket descriptor used by eSNMP.

subagent_identifier

The address of a null-terminated string that uniquely identifies this subagent (usually a program name).

Description

Call this routine during program initialization or to restart the eSNMP protocol. If you are restarting, the `esnmp_init` routine clears all registrations so each subtree must be registered again.

You should attempt to create a unique subagent identifier, perhaps using the program name `argv[0]` and additional descriptive text. The master agent does not open communications with a subagent whose subagent identifier is already in use.

This routine does not block waiting for a response from the master agent. After calling the `esnmp_init` routine, call the `esnmp_register` routine for each subtree that is to be handled by the subagent.

Return Values

ESNMP_LIB_NO_CONNECTION	Could not initialize or communicate with the master agent. Try again after a delay.
ESNMP_LIB_OK	The `esnmp_init` routine has completed successfully.
ESNMP_LIB_NOTOK	Could not allocate memory for the subagent.

Example

```
#include <esnmp_h>
int socket;
status = esnmp_init(&socket, "gated");
```

esnmp_register

esnmp_register — Requests local registration of a single MIB subtree. This indicates to the master agent that the subagent instantiates MIB variables within the registered MIB subtree.

Syntax

```
int esnmp_register ( subtree *subtree, int timeout, int priority );
```

Arguments

subtree

A pointer to a subtree structure corresponding to the subtree to be handled. The code emitted by the MIB compiler files (*subtree*_TBL.C and *subtree*_TBL.H) externally declare and initialize the subtree structures. Refer to Chapter 3 for more information about these files.

Note

All memory pointed to by the subtree fields must have permanent storage since it is referenced by `libesnmp` for the duration of the program. You should use the data declarations emitted by the MIBCOMP program.

timeout

The number of seconds the master agent should wait for responses when requesting data in this subtree. This value must be between 0 (zero) and 300. If the value is 0, the default timeout is 3 seconds.

VSI recommends that you use the default. For information about modifying the default subagent timeout value, refer to Section 6.2.

priority

The registration priority. The priority argument allows you to coordinate cooperating subagents to handle different configurations. The range is 1 to 255.

The entry with the largest number has the highest priority. The subagent that registers a subtree with the highest priority over a range of object identifiers gets all requests for that range of `OIDs`.

Subtrees registered with the same priority are considered duplicate, and the registration is rejected by the master agent.

Description

Call the initialization routine `esnmp_init` prior to calling the `esnmp_register`. Call the `esnmp_register` function for each subtree structure corresponding to each subtree to be handled. At any time, subtrees can be unregistered by calling `esnmp_unregister` and then be reregistered by calling the `esnmp_register`.

When restarting the eSNMP protocol by calling `esnmp_init`, all registrations are cleared. All subtrees must be reregistered.

A subtree is identified by the base MIB name and the corresponding `OID` number of the node that is the parent of all MIB variables contained in the subtree. For example: The MIB II `tcp` subtree has an `OID` of `1.3.6.1.2.1.6`. All elements subordinate to this have the same first seven digits and are included in the subtree's object table. The subtree can also be a single MIB object (a leaf node) or even a specific instance.

By registering a subtree, the subagent indicates that it will process eSNMP requests for all MIB variables (or `OIDs`) within that subtree's range. Therefore, a subagent should register the most fully qualified (longest) subtree that still contains its instrumented MIB variables.

The master agent does not permit a subagent to register the same subtree more than once. However, subagents can register subtrees with ranges that overlap the `OID` ranges of subtrees previously registered, and subagents can also register subtrees registered by other subagents.

For example, TCP/IP Services supports MIB II. In the eSNMP environment, the `os_mibs` subagent registers the MIB II subtree `ip` (OID 1.3.6.1.2.1.4).

TCP/IP Services also provides the `gated` subagent, which registers the `ipRouteEntry` MIB subtree (OID1.3.6.1.2.1.4.21.1).

These MIBs are registered at priority 1. Any subagent that registers at a higher priority (greater than 1) overrides these registrations.

A request for `IpRouteIfIndex` (OID1.3.5.1.2.1.4.21.1.2) is passed to the `gated` subagent. Requests for other `ip` variables, such as `ipNetToMediaIfIndex` (OID 1.3.5.1.2.1.4.22.1.1) are passed to the `os_mibs` subagent. If the `gated` subagent terminates or unregisters the `ipRouteEntry` subtree, subsequent requests for `ipRouteIfIndex` will go to the `os_mibs` subagent. This occurs because the `ip` subtree, which includes all `ipRouteEntry` variables, is now the authoritative region of requests for `ipRouteIfIndex`.

Return Values

SNMP_LIB_OK	The `esnmp_register` routine has completed successfully.
ESNMP_LIB_BAD_REG	The `esnmp_init` routine has not been called, the timeout parameter is invalid, or the subtree has already been queued for registration.
ESNMP_LIB_LOST_CONNECTION	The subagent has lost communications with the master agent.

Note that the return value indicates only the initiation of the request. The actual status returned in the master agent's response will be returned in a subsequent call to the `esnmp_poll` routine in the `state` field.

Example

```
#include <esnmp.h>
#define RESPONSE_TIMEOUT      0     /* use the default time set
                                       in OPEN message */
#define REGISTRATION_PRIORITY 10  /* priority at which subtrees
                                       will register */

int status;

extern SUBTREE ipRouteEntry_subtree;

status = esnmp_register( &ipRouteEntry_subtree,
                         RESPONSE_TIMEOUT,
                         REGISTRATION_PRIORITY );
if (status != ESNMP_LIB_OK) {
    printf ("Could not queue the 'ipRouteEntry' \n");
    printf ("subtree for registration\n");
}
```

esnmp_unregister

esnmp_unregister — Cancels registration of a MIB subtree previously registered with the master agent.

Syntax

```
int esnmp_unregister ( SUBTREE *subtree ) ;
```

Arguments

subtree

A pointer to a subtree structure corresponding to the subtree to be handled. The code emitted by the MIB compiler files (*subtree*_TBL.C and *subtree*_TBL.H) externally declare and initialize the subtree structures. Refer to Chapter 3 for more information about these files.

Description

This routine can be called by the application code to tell the eSNMP subagent not to process requests for variables in this MIB subtree anymore. You can later reregister a MIB subtree, if needed, by calling the esnmp_register routine.

Return Values

SNMP_LIB_OK	The esnmp_unregister routine has completed successfully.
ESNMP_LIB_BAD_REG	The MIB subtree was not registered.
ESNMP_LIB_LOST_CONNECTION	The request to unregister the MIB subtree could not be sent. You should restart the protocol.

Example

```
#include <esnmp.h>
int status

extern SUBTREE ipRouteEntry_subtree;

status = esnmp_unregister (&ipRouteEntry_subtree);

switch (status) {
case ESNMP_LIB_OK:
  printf ("The esnmp_unregister routine completed successfully.\n");
  break;

case ESNMP_LIB_BAD_REG:
  printf ("The MIB subtree was not registered.\n");

case ESNMP_LIB_LOST_CONNECTION:
  printf ("%s%s%s\n", "The request to unregister the ",
                      "MIB subtree could not be sent. ",
                      "You should restart the protocol.\n");

break;
}
```

esnmp_register2

esnmp_register2 — Requests registration of a single MIB subtree. This indicates to the master agent that the subagent instantiates MIB variables within the registered MIB subtree. The esnmp_register2 routine offers extensions to the esnmp_register routine.

Syntax

```
int esnmp_register2 ( ESNMP_REG *reg ) ;
```

Arguments

reg

A pointer to an ESNMP_REG structure that contains the following fields:

Field Name	Description
subtree	A pointer to a subtree structure corresponding to the MIB subtree to be handled. The subtree structures are externally declared and initialized in the code emitted by the MIBCOMP command (*subtree*_TBL.C and *subtree*_TBL.H, where *subtree* is the name of the MIB subtree). This code is taken directly from the MIB document. All memory pointed to by this field must have permanent storage since it is referenced by libesnmp for the duration of the program. You should use the data declarations emitted by the MIBCOMP command.
priority	The registration priority. The entry with the largest number has the highest priority. The range is 1 to 255. The subagent that has registered a MIB subtree with the highest priority over a range of object identifiers gets all requests for that range of OIDs. MIB subtrees that are registered with the same priority are considered duplicates, and the registration is rejected by the master agent. The priority field is a mechanism for cooperating subagents to handle different configurations.
timeout	The number of seconds the master agent should wait for responses when requesting data in this MIB subtree. This value must be between zero and 300. If the value is zero, the default timeout (3 seconds) is used. You should use the default. For information about modifying the default timeout value, refer to Section 6.2.
range_subid	An integer value that, when nonzero, together with the *range_upper_bound* field specifies a range instead of one of the MIB subtree's OID subidentifiers. The *range_subid* field specifies the OID subidentifier modified by the *range_upper_bound* field.
range_upper_bound	An integer value that, with a nonzero *range_subid* field, specifies a range instead of one of the MIB subtree's OID subidentifiers. The *range_upper_bound* field provides the upper bound of the range and the *range_subid* field provides the lower bound of the range, which is the MIB subtree's OID subidentifier.
options	An integer value that, when set to ESNMP_REG_OPT_CLUSTER, indicates that the registration is valid clusterwide. When the value is set to zero, it indicates that the registration is valid for the local node.

Field Name	Description	
state	One of the following integer values that provides the caller with asynchronous updates of the state of registration of this MIB subtree. After the return of the `esnmp_poll` routine, the caller can inspect this parameter.	
	ESNMP_REG_STATE _PENDING	The registration is currently held locally while waiting for connection to the master agent.
	ESNMP_REG_STATE_SENT	The registration was sent to the master agent.
	ESNMP_REG_STATE_DONE	The registration was successfully acknowledged by the master agent.
	ESNMP_REG_STATE _REGDUP	The registration was rejected by the master agent because it was a duplicate.
	ESNMP_REG_STATE _REGNOCLU	The master agent does not support cluster registrations.
	ESNMP_REG_STATE_REJ	The master agent rejected the registration for other reasons.
reserved	This field is reserved for exclusive use by the eSNMP library. The caller should not modify it.	

Description

The initialization routine (`esnmp_init`) must be called prior to calling the `esnmp_register2` routine. The `esnmp_register2` function must be called for each subtree structure corresponding to each MIB subtree that it will be handling. At any time, MIB subtrees can be unregistered by calling `esnmp_unregister2` and then can be reregistered by calling `esnmp_register2`.

When restarting the eSNMP protocol by calling `esnmp_init`, all MIB subtree registrations are cleared. All MIB subtrees must be reregistered.

A MIB subtree is identified by the base MIB variable name and its corresponding OID. This tuple represents the parent of all MIB variables that are contained in the MIB subtree; for example, the MIB II `tcp` subtree has an OID of 1.3.6.1.2.1.6. All elements subordinate to this (those that have the same first 7 identifiers) are included in the subtree's object table. A MIB subtree can also be a single MIB object (a leaf node) or even a specific instance.

By registering a MIB subtree, the subagent indicates that it will process SNMP requests for all MIB variables (or OIDs) within that MIB subtree's region. Therefore, a subagent should register the most fully qualified (longest) MIB subtree that still contains its instrumented MIB variables.

A subagent using the `esnmp_register2` routine can register the same MIB subtree for the local node and for a cluster. To register the MIB subtree for both, you must call the `esnmp_register2` routine twice: once with the ESNMP_REG_OPT_CLUSTER bit set in the *options* field and once with the ESNMP_REG_OPT_CLUSTER bit clear in the *options* field. Alternatively, you can register a MIB subtree for the cluster only or for the local node only, by setting or clearing the ESNMP_REG_OPT_CLUSTER bit, respectively, in the *options* field.

A subagent can also register MIB subtrees that overlap the OID range of MIB subtrees that it previously registered or the OID ranges of MIB subtrees registered by other subagents.

For example, consider the two subagents os_mibs and gated. The os_mibs subagent registers the ip MIB subtree (1.3.6.1.2.1.4), and the gated subagent registers the ipRouteEntry MIB subtree (1.3.6.1.2.1.4.21.1). Both of these registrations are made with the ESNMP_REG_OPT_CLUSTER bit set in the *options* field. Requests for ip MIB variables within ipRouteEntry, such as ipRouteIfIndex (1.3.6.1.2.1.4.21.1.2), are passed to the gated subagent. Requests for other ip variables, such as ipNetToMediaIfIndex (1.3.6.1.2.1.4.22.1.1), are passed to the os_mibs subagent. If the gated subagent terminates or unregisters the ipRouteEntry MIB subtree, subsequent requests for ipRouteIfIndex go to the os_mibs subagent. This occurs because the ip MIB subtree, which includes all ipRouteEntry MIB variables, is now the authoritative region of requests for ipRouteIfIndex.

Return Values

SNMP_LIB_OK	The esnmp_register2 routine has completed successfully.
ESNMP_LIB_BAD_REG	The esnmp_init routine has not been called, the timeout parameter is invalid, a registration slot is not available, or this MIB subtree has already been queued for registration. A message is also in the log file.
ESNMP_LIB_LOST_CONNECTION	The subagent lost communication with the master agent.

Note that the return value indicates only the initiation of the request. The actual status returned in the master agent's response will be returned in a subsequent call to the esnmp_poll routine in the state field.

Example

```
#include <esnmp.h>
#define RESPONSE_TIMEOUT      0    /* use the default time set
                                      in OPEN message */
#define REGISTRATION_PRIORITY 10  /* priority at which subtrees
                                      will register */

int status;

extern SUBTREE ipRouteEntry_subtree;

status = esnmp_register( &ipRouteEntry_subtree,
                         RESPONSE_TIMEOUT,
                         REGISTRATION_PRIORITY );
if (status != ESNMP_LIB_OK) {
    printf ("Could not queue the 'ipRouteEntry' \n");
    printf ("subtree for registration\n");
}
```

esnmp_unregister2

esnmp_unregister2 — Cancels registration of a MIB subtree previously established with the master agent. Use this routine only when the MIB subtree was registered using the `esnmp_register2` routine.

Syntax

```
int esnmp_unregister2 ( ESNMP_REG *reg ) ;
```

Arguments

reg

A pointer to the ESNMP_REG structure that was used when the `esnmp_register2` routine was called.

Description

This routine can be called by the application code to tell the eSNMP subagent to no longer process requests for variables in this MIB subtree. You can later reregister a MIB subtree, if needed, by calling the `esnmp_register2` routine.

Return Values

ESNMP_LIB_OK	The routine completed successfully.
ESNMP_LIB_BAD_REG	The MIB subtree was not registered.
ESNMP_LIB_LOST_CONNECTION	The request to unregister the MIB subtree could not be sent. You should restart the protocol.

Example

```
#include <esnmp.h>
int status

extern ESNMP_REG esnmp_reg_for_ip2egp;

status = esnmp_unregister2( &esnmp_reg_for_ip2egp );

switch(status) {
   case ESNMP_LIB_OK:
   printf("The esnmp_unregister2 routine completed successfully.\n");
   break;

   case ESNMP_LIB_BAD_REG:
   printf("The MIB subtree was not registered.\n");
   break;

   case ESNMP_LIB_LOST_CONNECTION:
   printf("%s%s%s\n", "The request to unregister the ",
                      "MIB subtree could not be sent.  ",
                      "You should restart the protocol.\n");
   break;
}
```

esnmp_capabilities

esnmp_capabilities — Adds a subagent's capabilities to the master agent's `sysORTable`. The `sysORTable` is a conceptual table that contains an agent's object resources, and is described in RFC 1907.

Syntax

```
void esnmp_capabilities ( OID *agent_cap_id,
                          char *agent_cap_descr ) ;
```

Arguments

agent_cap_id

A pointer to an object identifier that represents an authoritative agent capabilities identifier. This value is used for the `sysORID` object in the `sysORTable` for the managed node.

agent_cap_descr

A pointer to a null-terminated character string describing `agent_cap_id`. This value is used for the `sysORDescr` object in the `sysORTable` for the managed node.

Description

This routine is called at any point after initializing eSNMP by a call to the `esnmp_init` routine.

esnmp_uncapabilities

esnmp_uncapabilities — Removes a subagent's capabilities from the master agent's `sysORTable`.

Syntax

```
void esnmp_uncapabilities ( OID *agent_cap_id ) ;
```

Arguments

agent_cap_id

A pointer to an object identifier that represents an authoritative agent capabilities identifier. This value is used for the `sysORID` object in the `sysORTable` for the managed node.

Description

This routine is called if a subagent alters its capabilities dynamically. When a logical connection for a subagent is closed, the master agent automatically removes the related entries in `sysORTable`.

esnmp_poll

esnmp_poll — Processes a pending message that was sent by the master agent.

Syntax

```
int esnmp_poll ( )
```

Description

This routine is called after the `select()` call has indicated data is ready on the eSNMP socket. (This socket was returned from the call to the `esnmp_init` routine.)

If a received message indicates a problem, an entry is made to the SNMP log file and an error status is returned.

If the received message is a request for SNMP data, the object table is checked and the appropriate method routines are called, as defined by the developer of the subagent.

Return Values

ESNMP_LIB_OK	The `esnmp_poll` routine completed successfully.
ESNMP_LIB_BAD_REG	The master agent failed in a previous registration attempt. See the log file.
ESNMP_LIB_DUPLICATE	A duplicate subagent identifier has already been received by the master agent.
ESNMP_LIB_NO_CONNECTION	The master agent's `OPEN` request failed. Restart the connection after a delay. See the log file.
ESNMP_LIB_CLOSE	A `CLOSE` message was received.
ESNMP_LIB_NOTOK	An eSNMP protocol error occurred and the packet was discarded.
ESNMP_LIB_LOST_CONNECTION	Communication with the master agent was lost. Restart the connection.

esnmp_are_you_there

esnmp_are_you_there — Requests the master agent to report immediately that it is up and functioning.

Syntax

```
int esnmp_are_you_there ( ) ;
```

Description

The `esnmp_are_you_there` routine does not block waiting for a response. The routine is intended to cause the master agent to reply immediately. The response should be processed by calling the `esnmp_poll` routine.

If a response is not received within the timeout period, the application code should restart the eSNMP protocol by calling the `esnmp_init` routine. No timers are maintained by the eSNMP library.

Return Values

ESNMP_LIB_OK	The request was sent.
ESNMP_LIB_LOST_CONNECTION	The request cannot be sent because the master agent is down.

esnmp_trap

esnmp_trap — Sends a trap message to the master agent.

Syntax

```
int esnmp_trap ( int *generic_trap,
                 int specific_trap,
                 char *enterprise,
                 varbind *vb ) 2 ;
```

Arguments

generic_trap

A generic trap code. Set this argument value to 0 (zero) for SNMPv2 traps.

specific_trap

A specific trap code. Set this argument value to 0 (zero) for SNMPv2 traps.

enterprise

An enterprise OID string in dot notation. Set to the object identifier defined by the NOTIFICATION-TYPE macro in the defining MIB specification. This value is passed as the value of SnmpTrapOID.0 in the SNMPv2-Trap-PDU.

vb

A VARBIND list of data (a null pointer indicates no data).

Description

This function can be called any time. If the master agent is not running, traps are queued and sent when communication is possible.

The trap message is actually sent to the master agent after it responds to the `esnmp_init` routine. This occurs with every API call and for most `esnmp_register` routines. The quickest process to send traps to the master agent is to call the `esnmp_init, esnmp_poll,` and `esnmp_trap` routines.

The master agent formats the trap into an SNMP trap message and sends it to management stations based on its current configuration.

The master agent does not respond to the content of the trap. However, the master agent does return a value that indicates whether the trap was received successfully.

Return Values

ESNMP_LIB_OK	The routine has completed successfully.
ESNMP_LIB_LOST_CONNECTION	Could not send the trap message to master agent.
ESNMP_LIB_NOTOK	Something failed and the message could not be generated.

esnmp_term

esnmp_term — Sends a close message to the master agent and shuts down the subagent.

Syntax

```
void esnmp_term (void) ;
```

Description

Subagents must call this routine when terminating so that the master agent can update its MIB registry quickly and so that resources used by eSNMP on behalf of the subagent can be released.

Return Values

ESNMP_LIB_OK	The `esnmp_term` routine always returns ESNMP_LIB_OK, even if the packet could not be sent.

esnmp_sysuptime

esnmp_sysuptime — Converts UNIX system time obtained from `gettimeofday` into a value with the same time base as `sysUpTime`.

Syntax

```
unsigned int esnmp_sysuptime ( struct timeval *timestamp ) ;
```

Argument

timestamp

A pointer to `struct timeval`, which contains a value obtained from the `gettimeofday` system call. The structure is defined in `include/sys/time.h`.

A null pointer means return the current `sysUpTime`.

Description

This routine provides a mechanism to convert UNIX timestamps into eSNMP `TimeTicks`. The function returns the value that `sysUpTime` held when the passed timestamp was now.

This routine can be used as a `TimeTicks` data type (the time since the eSNMP master agent started) in hundredths of a second. The time base is obtained from the master agent in response to `esnmp_init`, so calls to this function before that time will not be accurate.

Return Values

0	An error occurred because a `gettimeofday` function failed. Otherwise, `timestamp` contains the time in hundredths of a second since the master agent started.

Example

```
#include <sys/time.h>
#include <esnmp.h>
struct timeval timestamp;

gettimeofday(&timestamp, NULL);
    .
    .
    .
o_integer(vb, object, esnmp_sysuptime(&timestamp));
```

5.2. Method Routines

SNMP requests may contain many encoded MIB variables. The `libsnmp` code executing in a subagent matches each `VariableBinding` with an object table entry. The object table's method routine is then called. Therefore, a method routine is called to service a single MIB variable. Since a single method routine can handle a number of MIB variables, the same method routine may be called several times during a single SNMP request.

The method routine calling interface contains the following functions:

- `*_get` – respond to `Get`, `GetNext`, and `GetBulk` requests

- `*_set` – respond to `Set` requests

*_get Routine

*_get Routine — The *_get routine is a method routine for the specified MIB item, which is typically a MIB group (for example, `system` in MIB II) or a table entry (for example, `ifEntry` in MIB II).

Syntax

```
int mib-group_get ( METHOD *method ) ;
```

Arguments

method

A pointer to a METHOD structure that contains the following fields:

Field Name	Description
action	One of ESNMP_ACT_GET, ESNMP_ACT_GETNEXT, or ESNMP_ACT_GETBULK.
serial_num	An integer number that is unique to this SNMP request. Each method routine called while servicing a single SNMP request receives the same value of *serial_num*. New SNMP requests are indicated by a new value of *serial_num*.
repeat_cnt	Used for `GetBulk` only. This value indicates the current iteration number of a repeating VARBIND. This number increments from 1 to

Field Name	Description
	max_repetitions and is 0 (zero) for non repeating VARBIND structures.
max_repetitions	The maximum number of repetitions to perform. Used for `GetBulk` only. This will be 0 (zero) for non-repeating VARBIND structures. You can optimize subsequent processing by knowing the maximum number repeat calls will be made.
varbind	A pointer to the VARBIND structure for which you must fill in the OID and data fields. Upon entry of the method routine, the *method->varbind->name* field is the OID that was requested. Upon exit of the method routine, the *method->varbind* field contains the requested data, and the *method->varbind->name* field is updated to reflect the actual instance OID for the returned VARBIND structure. The support routines (`o_integer`, `o_string`, `o_oid`, and `o_octet`) are generally used to load data. The libsnmp `instance2oid` routine is used to update the OID in the *method->varbind->name* field.
object	A pointer to the object table entry for the MIB variable being referenced. The *method->object->object_index* field is this object's unique index within the object table (useful when one method routine services many objects). The *method->object->oid* field is the OID defined for this object in the MIB. The instance requested is derived by comparing this OID with the OID in the request found in the *method->varbind->name* field. The `oid2instance` function is useful for this.

Description

These types of routines call whatever routine is specified for `Get` operations in the object table identified by the registered subtree.

This function is pointed to by some number of elements of the subagent object table. When a request arrives for an object, its method routine is called. The `*_get` method routine is called in response to a `Get` request.

Return Values

ESNMP_MTHD_noError	The routine completed successfully.
ESNMP_MTHD_noSuchObject	The requested object cannot be returned or does not exist.

ESNMP_MTHD_noSuchInstance	The requested instance of an object cannot be returned or does not exist.
ESNMP_MTHD_genErr	A general processing error.

*_set Routine

*_set Routine — The `*_set` method routine for a specified MIB item, which is typically a MIB group (for example, `system` in MIB II) or a table entry (for example, `ifEntry` in MIB II).

Syntax

```
int mib-group_set ( METHOD *method ) ;
```

Arguments

method

A pointer to a METHOD structure that contains the following fields:

Field Name	Description
action	One of ESNMP_ACT_SET, ESNMP_ACT_UNDO, or ESNMP_ACT_CLEANUP.
serial_num	An integer number that is unique to this SNMP request. Each method routine called while servicing a single SNMP request receives the same value as *serial_num*. New SNMP requests are indicated by a new value of *serial_num*.
varbind	A pointer to the VARBIND structure that contains the MIB variable's supplied data value and name (OID). The instance information has already been extracted from the OID and placed in the *method->row->instance* field.
object	A pointer to the object table entry for the MIB variable being referenced. The *method->object->object-index* field is this object's unique index within the object table (useful when one method routine services many objects). The *method->object->oid* field is the OID defined for this object in the MIB.
flags	A read-only integer bitmask set by the `libesnmp` routine. If set, the ESNMP_FIRST_IN_ROW bit indicates that this call is the first object to be set in the row. If set, the ESNMP_LAST_IN_ROW bit indicates that this call is the last object to be set in the row. Only METHOD structures with the ESNMP_LAST_IN_ROW bit set are passed to the method routines for commit, undo, and cleanup phases.
row	A pointer to a ROW_CONTEXT structure (defined in the ESNMP.H header file). All `Set` requests to the method routine that refer to the same group and that have the same instance number will be presented with the same row structure. The method routines can accumulate information in the row structures during `Set` requests for use during the commit and undo phases. The

Field Name	Description	
	accumulated data can be released by the method routines during the cleanup phase. The ROW_CONTEXT structure contains the following fields:	
	instance	An address of an array containing the instance OID for this conceptual row. The libesnmp routine builds this array by subtracting the object OID from the requested variable binding OID.
	instance_len	The size of the *method->row->instance* field.
	context	A pointer to be used privately by the method routine to reference data needed for processing this request.
	save	A pointer to be used privately by the method routine to reference data needed for undoing this request.
	state	An integer to be used privately by the method routine for holding any state information it requires.

Description

The libesnmp routines call whatever routine is specified for Set operations in the object table identified by the registered subtree.

This function is pointed to by some number of elements of the subagent object table. When a request arrives for an object, its method routine is called. The *_set method routine is called in response to a Set request.

Return Values

ESNMP_MTHD_noError	The routine completed successfully.
ESNMP_MTHD_notWritable	The requested object cannot be set or was not implemented.
ESNMP_MTHD_wrongType	The data type for the requested value is the wrong type.
ESNMP_MTHD_wrongLength	The requested value is the wrong length.
ESNMP_MTHD_wrongEncoding	The requested value is represented incorrectly.
ESNMP_MTHD_wrongValue	The requested value is out of range.
ESNMP_MTHD_noCreation	The requested instance can never be created.
ESNMP_MTHD_inconsistentName	The requested instance cannot currently be created.

ESNMP_MTHD_inconsistentValue	The requested value is not consistent.
ESNMP_MTHD_resourceUnavailable	A failure due to some resource constraint.
ESNMP_MTHD_genErr	A general processing error.
ESNMP_MTHD_commitFailed	The commit phase failed.
ESNMP_MTHD_undoFailed	The undo phase failed.

5.3. Processing *_set Routines

This following is the sequence of operations performed for `*_set` routines:

1. Every variable binding is parsed and its object is located in the object table. A METHOD structure is created for each VARBIND structure. These METHOD structures point to a ROW_CONTEXT structure, which is useful for handling these phases. Objects in the same conceptual row all point to the same ROW_CONTEXT structure. This determination is made by checking the following:

   - The referenced objects are in the same MIB group.

   - The VARBIND structures have the same instance OIDs.

2. Each ROW_CONTEXT structure is loaded with the instance information for that conceptual row. The ROW_CONTEXT structure context and save fields are set to NULL, and the state field is set to ESNMP_SET_UNKNOWN structure.

3. The method routine for each object is called and is passed its METHOD structure with an action code of ESNMP_ACT_SET.

   If all method routines return success, a single method routine (the last one called for the row) is called for each row, with *method->action* equal to ESNMP_ACT_COMMIT.

   If any row reports failure, all rows that were successfully committed are told to undo the phase. This is accomplished by calling a single method routine for each row (the same one that was called for the commit phase), with a *method->action* equal to ESNMP_ACT_UNDO.

4. Each row is released. The same single method routine for each row is called with a *method->action* equal to ESNMP_ACT_CLEANUP. This occurs for every row, regardless of the results of previous processing.

The action codes are processed as follows:

- ESNMP_ACT_SET

   Each object's method routine is called during the SET phase, until all objects are processed or a method routine returns an error status value. (This is the only phase during which each object's method routine is called.) For variable bindings in the same conceptual row, *method->row* points to a common ROW_CONTEXT.

   The *method->flags* bitmask has the ESNMP_LAST_IN_ROW bit set, if this is the last object being called for this ROW_CONTEXT. This enables you to do a final consistency check, because you have seen every variable binding for this conceptual row.

   The method routine's job in this phase is to determine whether the `Set` request will work, to return the correct SNMP error code if it does not, and to prepare any context data it needs to actually perform the `Set` request during the COMMIT phase.

The *method->row->context* field is private to the method routine; `libesnmp` does not use it. A typical use is to store the address of an emitted structure that has been loaded with the data from the VARBIND for the conceptual row.

- ESNMP_ACT_COMMIT

  Even though several variable bindings may be in a conceptual row, only the last one in order of the `Set` request is processed. Of all the method routines that point to a common row, only the last method routine is called.

  This method routine must have available to it all necessary data and context to perform the operation. It must also save a snapshot of current data or whatever it needs to undo the `Set` operation, if required. The *method->row->save* field is intended to hold a pointer to whatever data is needed to accomplish this. A typical use is to store the address of a structure that has been loaded with the current data for the conceptual row. The structure is one that has been automatically generated by the MIBCOMP command.

  The *method->row->save* field is also private to the method routine; `libesnmp` does not use it.

  If this operation succeeds, return `ESNMP_MTHD_noError`; otherwise, return a value of `ESNMP_MTHD_commitFailed`.

  If any errors were returned during the COMMIT phase, `libesnmp` enters the UNDO phase; if not, it enters the CLEANUP phase.

Note

If the `Set` request spans multiple subagents and another subagent fails, the UNDO phase may occur even if the `Set` operation is successful

---

- ESNMP_ACT_UNDO

  For each conceptual row that was successfully committed, the same method routine is called with *method->action* equal to ESNMP_ACT_UNDO. The ROW_CONTEXT structures that have not yet been called for the COMMIT phase are not called for the UNDO phase; they are called for CLEANUP phase.

  The method routine should attempt to restore conditions to what they were before it executed the COMMIT phase. (This is typically done using the data pointed to by the *method->row->save* field.)

  If successful, return ESNMP_MTHD_noError; otherwise, return ESNMP_MTHD_undoFail.

- ESNMP_ACT_CLEANUP

  Regardless of what else has happened, at this point each ROW_CONTEXT participates in cleanup phase. The same method routine that was called for in the COMMIT phase is called with *method->action* equal to ESNMP_ACT_CLEANUP.

  This indicates the end of processing for the `set` request. The method routine should perform whatever cleanup is required; for instance, freeing dynamic memory that might have been allocated and stored in *method->row->context* and *method->row->save* fields, and so on.

  The function return status value is ignored for the CLEANUP phase.

5.4. Method Routine Applications Programming

You must write the code for the method routines declared in the *subtree*_TBL.H file. Each method routine has one argument, which is a pointer to the METHOD structure, as follows:

```
int mib_group_get(
        METHOD *method int mib_group_set(
        METHOD *method );
```

The `Get` method routines are used to perform `Get`, `GetNext`, and `GetBulk` operations.

The `Get` method routines perform the following tasks:

- Extract the instance portion of the requested OID. You can do this manually by comparing the *method->object->oid* field (the object's base OID) to the *method-> varbind->name* field (the requested OID). You can use the `oid2instance` support routine to do this.

- Determine the instance validity. The instance OID can be null or any length, depending on what was requested and how your object was selected. You may be able to reject the request immediately by checking on the instance OID.

- Extract the data. Based on the instance OID and *method->action* field, determine what data, if any, is to be returned.

- Load the response OID back into the method routine's VARBIND structure. Set the *method->varbind* field with the OID of the actual MIB variable instance you are returning. This is usually accomplished by loading an array of integers with the instance OID you wish to return and calling the `instance2OID` support routine.

- Load the response data back into the method routine's VARBIND structure.

  Use one of the support routines with the corresponding datatype to load the *method->varbind* field with the data to return:

  - `o_integer`

  - `o_string`

  - `o_octet`

  - `o_oid`

  These routines make a copy of the data you specify. The `libesnmp` function manages any memory associated with copied data. The method routine must manage the original data's memory.

  The routine does any necessary conversions to the type defined in the object table for the MIB variable and copies the converted data into the *method->varbind* field. See Section 5.5 for information on data value representation.

- Return the correct status value, as follows:

ESNMP_MTHD_noError	The routine completed successfully or no errors were found.

ESNMP_MTHD_noSuchInstance	There is no such instance of the requested object.
ESNMP_MTHD_noSuchObject	No such object exists.
ESNMP_MTHD_ genErr	An error occurred and the routine did not complete successfully.

5.5. Value Representation

The values in a VARBIND structure for each data type are represented as follows. (Refer to the ESNMP.H file for a definition of the OCT and OID structures.)

- ESNMP_TYPE_Integer32 (*varbind->value.sl* field)

  This is a 32-bit signed integer. Use the `o_integer` routine to insert an integer value into the VARBIND structure. Note that the prototype for the value argument is unsigned long; therefore, you may need to cast this to a signed integer.

- ESNMP_TYPE_DisplayString, ESNMP_TYPE_Opaque

  ESNMP_TYPE_OctetString (*varbind->value.oct* field)

  This is an octet string. It is contained in the VARBIND structure as an OCT structure that contains a length and a pointer to a dynamically allocated character array.

  The *displaystring* is different only in that the character array can be interpreted as ASCII text, but the *octetstring* can be anything. If the *octetstring* contains bits or a bit string, the OCT structure contains the following:

  - A length equal to the number of bytes needed to contain the value that is ((*qty-bits* - 1)/8 + 1)

  - A pointer to a buffer containing the bits of the bitstring in the form *bbbbb..bb*, where the *bb* octets represent the bitstring itself, bit 0 comes first, and so on. Any unused bits in the last octet are set to zero.

  Use the `o_string` support routine to insert a value into the VARBIND structure, which is a buffer and a length. New space is allocated and the buffer is copied into the new space.

  Use the `o_octet` routine to insert a value into the VARBIND structure, which is a pointer to an OCT structure. New space is allocated and the buffer pointed to by the OCT structure is copied.

- ESNMP_TYPE_ObjectId (*varbind->value.oid* and the *varbind->name* fields)

  This is an object identifier. It is contained in the VARBIND structure as an OID structure that contains the number of elements and a pointer to a dynamically allocated array of unsigned integers, one for each element.

  The *varbind->name* field is used to hold the object identifier and the instance information that identifies the MIB variable. Use the `OID2Instance` function to extract the instance elements from an incoming OID on a request. Use the `instance2oid` function to combine the instance elements with the MIB variable's base OID to set the VARBIND structure's name field when building a response.

  Use the `o_oid` function to insert an object identifier into the VARBIND structure when the OID value to be returned as data is in the form of a pointer to an OID structure.

Use the `o_string` function to insert an OID into the VARBIND structure when the OID value to be returned as data is in the form of a pointer to an ASCII string containing the OID in dot format; for example: 1.3.6.1.2.1.3.1.1.2.0.

- ESNMP_TYPE_NULL

  This is the NULL, or empty, type. This is used to indicate that there is no value. The length is zero and the value in the VARBIND structure is zero filled.

  The incoming VARBIND structures on a `Get`, `GetNext`, and `GetBulk` will have this data type. A method routine should never return this value. An incoming `Set` request never has this value in a VARBIND structure.

- ESNMP_TYPE_IpAddress (*varbind->value.oct* field)

  This is an IP address. It is contained in the VARBIND structure in an OCT structure that has a length of 4 and a pointer to a dynamically allocated buffer containing the 4 bytes of the IP address in network byte order.

  Use the `o_integer` function to insert an IP address into the VARBIND structure when the value is an unsigned integer in network byte order.

  Use the `o_string` function to insert an IP address into the VARBIND structure when the value is a byte array (in network byte order). Use a length of 4.

- ESNMP_TYPE_Integer32

  ESNMP_TYPE_Counter32

  ESNMP_TYPE_<Gauge32 (*varbind->value.ul* field)

  The 32-bit counter and 32-bit gauge data types are stored in the VARBIND structure as an unsigned integer.

  Use the `o_integer` function to insert an unsigned value into the VARBIND structure.

- ESNMP_TYPE_TimeTicks (*varbind->value.ul* field)

  The 32-bit `timeticks` type values are stored in the VARBIND structure as an unsigned integer.

  Use the `o_integer` function to insert an unsigned value into the VARBIND structure.

- ESNMP_TYPE_Counter64 (*varbind->value.ul64* field)

  The 64-bit counter is stored in a VARBIND structure as an unsigned longword, which, on an OpenVMS Alpha system, has a 64-bit value.

  Use the `o_integer` function to insert an unsigned longword (64 bits) into the VARBIND structure.

5.6. Support Routines

The support routines are provided as a convenience for developers writing method routines that handle specific MIB elements. The following support routines are provided:

Routine	Function
`o_integer`	Loads an integer value.
`o_octet`	Loads an octet value.
`o_oid`	Loads an OID value.
`o_string`	Loads a string value.
`o_counter64`	Loads a Counter64 variable into the `varbind`.
`str2oid`	Converts a string OID to dot notation.
`sprintoid`	Converts an OID into a string.
`instance2oid`	Creates a full OID for a value.
`oid2instance`	Extracts an instance and loads an array.
`inst2ip`	Returns an IP address for an OID.
`cmp_oid`	Compares two OIDs.
`cmp_oid_prefix`	Compares an OID's prefix.
`clone_oid`	Makes a copy of an OID.
`free_oid`	Frees a buffer.
`clone_buf`	Duplicates a buffer.
`mem2oct`	Converts a string to an `oct` structure.
`cmp_oct`	Compares two octets.
`clone_oct`	Makes a copy of an `oct` structure.
`free_oct`	Frees a buffer attached to an `oct` structure.
`free_varbind_date`	Frees the fields in the `VARBIND` structure.
`set_debug_level`	Sets the logging level.
`is_debug_level`	Tests the logging level.
`ESNMP_LOG`	Directs log messages.
`print_varbind`	Displays the `varbind` and its structure.
`set_select_limit`	Sets the error limit for SNMP client requests.
`__set_progname`	Sets the program name to be displayed in log messages.
`__restore_progname`	Resets the program name back to the previous name.
`__parse_progname`	Parses the application file name to determine the program name.
`esnmp_cleanup`	Closes a socket that is used by a subagent for communicating with the master agent.

o_integer

o_integer — Loads an integer value into the VARBIND structure with the appropriate type. This function does not allocate the VARBIND structure.

Syntax

```
int o_integer ( VARBIND *vb,
                OBJECT *obj,
                unsigned long value );
```

Arguments

vb

A pointer to the VARBIND structure that is supposed to receive the data.

obj

A pointer to the `OBJECT` structure for the MIB variable associated with the `OID` in the VARBIND structure.

value

The value to be inserted into the VARBIND structure.

The real type as defined in the object structure must be one of the following; otherwise, an error is returned.

ESNMP_TYPE_Integer32	32-bit integer
ESNMP_TYPE_Counter32	32-bit counter (unsigned)
ESNMP_TYPE_Gauge32	32-bit gauge (unsigned)
ESNMP_TYPE_TimeTicks	32-bit timeticks (unsigned)
ESNMP_TYPE_UInteger32	32-bit integer (unsigned)
ESNMP_TYPE_Counter64	64-bit counter (unsigned)
ESNMP_TYPE_IpAddress	Implicit octet string (4)

Note

If the real type is `IpAddress`, then eSNMP assumes that the 4-byte integer is in network byte order and packages it into an octet string.

Return Values

ESNMP_MTHD_noError	The routine completed successfully.
ESNMP_MTHD_genErr	An error has occurred.

Example

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND    *vb      = method->varbind;
OBJECT     *object  = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
```

```
    :
switch(arg) {
case I_atIfIndex:
return o_integer(vb, object, data->ipNetToMediaIfIndex);
```

o_octet

o_octet — Loads an octet value into the VARBIND structure with the appropriate type. This function does not allocate the VARBIND structure.

Syntax

```
int o_octet ( VARBIND *vb,
              OBJECT *obj,
              unsigned long value );
```

Arguments

vb

A pointer to the VARBIND structure that is supposed to receive the data.

If the original value in the *vb* field is not null, this routine attempts to free it. So if you dynamically allocate memory or issue the `malloc` command to allocate your own VARBIND structure, fill the structure with zeros before using it.

obj

A pointer to the OBJECT structure for the MIB variable associated with the OID in the VARBIND structure.

value

The value to be inserted into the VARBIND structure.

The real type as defined in the object structure must be one of the following; otherwise, an error is returned.

ESNMP_TYPE_OCTET_STRING	Octet string (ASN.1)
ESNMP_TYPE_IpAddress	Implicit octet string (4) (in octet form, network byte order)
ESNMP_TYPE_DisplayString	DisplayString (textual convention)
ESNMP_TYPE_Opaque	Implicit octet string

Return Values

ESNMP_MTHD_noError	The routine completed successfully.
ESNMP_MTHD_genErr	An error occurred.

Example

```
#include <esnmp.h>
```

```
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND      *vb       = method->varbind;
OBJECT       *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
 case I_atPhysAddress:
 return o_octet(vb, object, &data->ipNetToMediaPhysAddress);
```

o_oid

o_oid — Loads an OID value into the VARBIND structure with the appropriate type. This function does not allocate the VARBIND structure.

Syntax

```
int o_oid ( VARBIND *vb,
          OBJECT *obj,
          OID *oid );
```

Arguments

vb

A pointer to the VARBIND structure that is supposed to receive the data.

If the original value in the VARBIND structure is not null, this routine attempts to free it. So if you dynamically allocate memory or issue the malloc command to allocate your own VARBIND structure, fill the structure with zeros before using it.

obj

A pointer to the OBJECT structure for the MIB variable associated with the OID in the VARBIND structure.

oid

The value to be inserted into the VARBIND structure as data. For more information about OID length and values, see Chapter 3.

The real type as defined in the object structure must be ESNMP_TYPE_OBJECT_IDENTIFIER.

Return Values

ESNMP_MTHD_noError	The routine completed successfully.
ESNMP_MTHD_genErr	An error occurred.

Example

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND      *vb       = method->varbind;
```

```
OBJECT      *object  = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
 case I_atObjectID:
   return o_oid(vb, object, &data->ipNetToMediaObjectID);
```

o_string

o_string — Loads a string value into the VARBIND structure with the appropriate type. This function does not allocate the VARBIND structure.

Syntax

```
int o_string ( VARBIND *vb,
             OBJECT *obj,
             unsigned character *ptr,
             int len );
```

Arguments

vb

A pointer to the VARBIND structure that is supposed to receive the data.

If the original value in the VARBIND structure is not null, this routine attempts to free it. So if you dynamically allocate memory or issue the `malloc` command to allocate your own VARBIND structure, fill the structure with zeros before using it.

obj

A pointer to the OBJECT structure for the MIB variable associated with the `OID` in the VARBIND structure.

ptr

The pointer to the buffer containing data to be inserted into the VARBIND structure as data.

len

The length of the data in buffer pointed to by *ptr*.

The real type as defined in the object structure must be one of the following; otherwise, an error is returned.

ESNMP_TYPE_OCTET_STRING	Octet string (ASN.1)
ESNMP_TYPE_IpAddress	Implicit octet string (4) (in octet form, network byte order)
ESNMP_TYPE_DisplayString	DisplayString (textual convention)
ESNMP_TYPE_NsapAddress	Implicit octet string
ESNMP_TYPE_Opaque	Implicit octet string

ESNMP_TYPE_OBJECT_IDENTIFIER	Object identifier (ASN.1) (in dot notation, for example: 1.3.4.6.3)

Return Values

ESNMP_MTHD_noError	The routine completed successfully.
ESNMP_MTHD_genErr	An error occurred.

Example

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND    *vb       = method->varbind;
OBJECT     *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
 case I_atPhysAddress:
   return o_string(vb, object, data->ipNetToMediaPhysAddress.ptr,
                              data->ipNetToMediaPhysAddress.len);
```

o_counter64

o_counter64 — Loads a counter64 value into the VARBIND structure.

Syntax

```
int o_counter64 ( VARBIND *vb,
              OBJECT *obj,
              uint64 value ); (for Alpha)
              uint64_vax value ; (for VAX))
```

Arguments

vb

A pointer to the VARBIND structure that is supposed to receive the data.

obj

A pointer to the OBJECT structure for the MIB variable associated with the OID in the VARBIND structure.

value

The 8-byte value to be inserted into the VARBIND structure, passed as an array of two integers.

The real type as defined in the object structure must be ESNMP_TYPE_Counter64. Otherwise, an error is returned.

Example

See the example for the o_integer routine.

Return Values

ESNMP_MTHD_noError	No error was generated.
ESNMP_MTHD_genErr	An error was generated.

str2oid

str2oid — Converts a null-terminated string `OID` in dot notation to an `OID` structure. The `str2oid` routine does not allocate an `OID` structure.

Syntax

```
oid *str2oid ( oid *oid,
               char *s );
```

Arguments

oid

The value to be inserted as data into the VARBIND structure. For more information about OID length and values, see Chapter 3.

s

A null string or empty string returns an OID structure that has one element of zero.

Description

The routine dynamically allocates the buffer and inserts its pointer into the `OID` structure passed in the call. The caller must explicitly free this buffer. The `OID` can have a maximum of 128 elements.

Return Values

null	An error occurred. Otherwise, the pointer to the `OID` structure (its first argument) is returned.

Example

```
include <esnmp.h>
OID abc;
if (stroid (&abc, "1.2.5.4.3.6") == NULL
   DPRINTF((WARNING, "It did not work...\n");
```

sprintoid

sprintoid — Converts an `OID` into a null-terminated string.

Syntax

```
char *sprintoid ( char *buffer, oid *oid );
```

Description

An `OID` can have up to 128 elements. A full-sized `OID` can require a large buffer.

Return Values

The return value points to its first argument.

Example

```
#include <esnmp.h>
#define SOMETHING_BIG 1024
OID abc;
char buffer[SOMETHING_BIG];
:
: assume abc gets initialized with some value
:
printf("dots=%s\n", sprintoid(buffer, &abc));
```

instance2oid

instance2oid — Copies the object's base OID and appends a copy of the instance array to make a complete OID for a value. This routine does not allocate an OID structure. It only allocates the array containing the elements.

Syntax

```
oid instance2oid ( oid *new,
                   object *obj,
                   unsigned int *instance,
                   int *len );
```

Arguments

new

A pointer to the OID that is to receive the new OID value.

obj

A pointer to the object table entry for the MIB variable being obtained. The first part of the new OID is the OID from this MIB object table entry.

instance

A pointer to an array of `instance` values. These values are appended to the base OID obtained from the MIB object table entry to construct the new OID.

len

The number of elements in the `instance` array.

Description

The instance array may be created by `oid2instance` or constructed from key values as a result of a `GetNext` command (see Chapter 1).

This routine dynamically allocates the buffer and inserts its pointer into the OID structure passed in the call. The caller must explicitly free the buffer.

You should point to the OID structure receiving the new values and then call the `instance2oid` routine. Previous values in the OID structure are freed (that is, `free_oid` is called first), and then the new values are dynamically allocated and inserted. Be sure the initial value of the new OID is all zeros. If you do not want the initial value freed, make sure the new OID structure is all zeros.

Return Values

null	An error occurred. Otherwise, the pointer to the OID structure (*new*) is returned.

Example

```
#include <esnmp.h>
VARBIND *vb;       <-- filled in
OBJECT  *object;  <-- filled in
unsigned int instance[6];

-- Construct the outgoing OID in a GETNEXT          --
-- Instance is N.1.A.A.A.A where A's are IP address --
instance[0] = data->ipNetToMediaIfIndex;
instance[1] = 1;
for (i = 0; i < 4; i++) {
instance[i+2]=((unsigned char *)(&data->ipNetToMediaNetAddress))[i];
}
instance2oid(&vb->name, object, instance, 6);
```

oid2instance

oid2instance — Extracts the instance values from an OID structure and copies them to the specified array of integers. The routine then returns the number of elements in the array.

Syntax

```
int oid2instance ( oid *oid,
                  object *obj,
                  unsigned int *instance,
                  int *max_len );
```

Arguments

oid

A pointer to an incoming OID containing an instance or part of an instance.

obj

A pointer to the object table entry for the MIB variable.

instance

A pointer to an array of unsigned integers where the index is placed.

max_len

The number of elements in the `instance` array.

Description

The instance values are the elements of an OID beyond those that identify the MIB variable. These elements identify a specific instance of a MIB value.

If there are more elements in the OID structure than specified by the `max_len` parameter, the function copies the number of elements specified by `max_len` only and returns the total number of elements that would have been copied had there been space.

Return Values

Less than zero	An error occurred. This is not returned if the object was obtained by looking at this OID.
Zero	No instance elements.
Greater than zero	The returned value indicates the number of elements in the index. This could be larger than the `max_len` parameter.

Example

```
#include <esnmp.h>
OID          *incoming = &method->varbind->name;
OBJECT       *object   = method->object;
int          instLength;
unsigned int instance[6];

-- in a GET operation --
-- Expected Instance is N.1.A.A.A.A where A's are IP address --
instLength = oid2instance(incoming, object, instance, 6);
if (instLength != 6)
    return ESNMP_MTHD_noSuchInstance;
```

The N will be in `instance[0]` and the IP address will be in `instance[2]`, `instance[3]`, `instance[4]`, and `instance[5]`.

inst2ip

inst2ip — Returns an IP address derived from an OID instance.

Syntax

```
int inst2ip ( unsigned int *instance,
          int *length,
          unsigned int *ipaddr,
          int *exact,
          int *carry );
```

Arguments

instance

A pointer to an array of `unsigned int` containing the instance numbers returned by the `oid2instance` routine to be converted to an IP address.

The range for elements is between zero and 255. Using the EXACT mode, the routine returns 1 if an element is out of range. Using the NEXT mode, a value greater than 255 causes that element to overflow. In this case, the value is set to 0 and the next most significant element is incremented; therefore, it returns a lexically equivalent value of the next possible *ipaddr*.

length

The number of elements in the instance array. Instances beyond the fourth are ignored. If the length is less than four, the missing values are assumed to be zero. A negative length results in an *ipaddr* value of zero. For an exact match (such as `Get`), there must be exactly four elements.

ipAddr

A pointer indicating where to return the IP address value. This routine is in network byte order (the most significant element is first).

exact

Can either be `TRUE` or `FALSE`.

`TRUE` means do an EXACT match. If any element is greater than 255 or if there are not exactly four elements, a value of 1 is returned. The carry argument is ignored.

`FALSE` means do a NEXT match. That is, the lexically next IP address is returned, if the carry value is set and the length is at least four. If there are fewer than four elements, this function assumes the missing values are zero. If any one element contains a value greater than 255, the value is zeroed and the next most significant element is incremented. Returns a 1 (one) only when there is a carry from the most significant (the first) value.

carry

Adds to the IP address on a NEXT match. If you are trying to determine the next possible IP address, pass in a one. Otherwise, pass in a zero. A length of less than 4 cancels the carry.

Description

Use the EXACT mode for evaluating an instance for `Get` and `Set` operations. For `GetNext` and `GetBulk` operations, use the NEXT mode. When using NEXT mode, this routine assumes that the search for data will be performed using greater than or equal to matches.

Return Values

Carry value is 0	The routine completed successfully.
Carry value is 1	For EXACT match, an error occurred. For NEXT match, there was a carry. If there was a carry, the returned *ipaddr* is 0.

Examples

1. The following example converts an instance to an IP address for a `Get` operation, which is an EXACT match.

   ```
   #include <esnmp.h>
   ```

```
OID          *incoming = &method->varbind->name;
OBJECT       *object   = method->object;
int instLength;
unsigned int instance[6];
unsigned int ip_addr;
int          iface;

-- The instance is N.1.A.A.A.A where the A's are the IP address--
instLength = oid2instance(incoming, object, instance, 6);
if (instLength == 6 && !inst2ip(&instance[2], 4, &ip_addr, TRUE, 0)) {
    iface = (int) instance[0];
}
else
    return ESNMP_MTHD_noSuchInstance;
```

2. The following example shows a `GetNext` operation in which there is only one key or in which the *ipaddr* value is the least significant part of the key. This is a NEXT match; therefore, a value of 1 is passed back for the carry value.

```
#include <esnmp.h>
OID          *incoming = &method->varbind->name;
OBJECT       *object   = method->object;
int instLength;
unsigned int instance[6];
unsigned int ip_addr;
int          iface;

-- The instance is N.1.A.A.A.A where the A's are the IP address--
instLength = oid2instance(incoming, object, instance, 6);
iface = (instLength < 1) ? 0 :(int) instance[0];

iface += inst2ip(&instance[2], instLength - 2, &ip_addr, FALSE, 1);
```

3. In the following example, the search key consists of multiple parts. If you are doing a `GetNext` operation, you must find the next possible value for the search key, so that you can perform a cascaded greater-than or equal-to search.

The search key consists of a number and two *ipaddr* values. These are represented in the instance part of the OID as *n*.A.A.A.A.B.B.B.B, where:

- *n* is a single value integer.

- The A.A.A.A portion makes up one IP address.

- The B.B.B.B portion makes up a second IP address.

If all elements are given, the total length of the search key is 9. In this case, you perform the operation as follows:

- Convert the least significant part of the key (that is, the B.B.B.B portion), by calling the `inst2ip` routine, passing it a 1for the carry and (*length* - 5) for the length.

- If the conversion of the B.B.B.B portion generates a carry (that is, returns 1), you pass it to the next most significant part of the key.

- Convert the A.A.A.A portion by calling the `inst2ip` routine, passing it (*length* - 1) for the length and the carry returned from the conversion of the B.B.B.B portion.

- The most significant element *n* is a number; therefore, add the carry from the A.A.A.A conversion to the number. If the result overflows, then the search key is not valid.

```
#include <esnmp.h>
OID         *incoming = &method->varbind->name;
OBJECT      *object   = method->object;
int instLength;
unsigned int instance[9];
unsigned int ip_addrA;
unsigned int ip_addrB;
int         iface;
int         carry;

-- The instance is N.A.A.A.B.B.B.B --
instLength = oid2instance(incoming, object, instance, 9);
iface = (instLength < 1) ? 0 :(int) instance[0];
carry = inst2ip(&instance[1], instLength - 1, &ip_addrB, FALSE, 1);
carry = inst2ip(&instance[5], instLength - 5, &ip_addrA, FALSE, carry);
iface += carry;
if (iface > carry) {

-- a carry caused an overflow in the most significant element
return ESNMP_MTHD_noSuchInstance;
}
```

cmp_oid

cmp_oid — Compares two OID structures.

Syntax

```
int cmp_oid ( oid *q, oid *p );
```

Description

This routine does an element-by-element comparison, from the most significant element (element 0) to the least significant element. If all other elements are equal, the OID with the least number of elements is considered less.

Return Values

-1	The OID *q* is less than OID *p*.
0	The OID *q* is in OID *p*.
1	The OID *q* is greater than OID *p*.

cmp_oid_prefix

cmp_oid_prefix — Compares an OID against a prefix.

Syntax

```
int cmp_oid_prefix ( oid *q, oid *prefix );
```

Description

A prefix could be the OID on an object in the object table. The elements beyond the prefix are the instance information.

This routine does an element-by-element comparison, from the most significant element (element 0) to the least significant element. If all elements of the prefix OID match exactly with corresponding elements of the `OID q` structure, it is considered an even match if the `OID q` structure contains additional elements. The `OID q` structure is considered greater than the prefix if the first nonmatching element is larger. It is considered smaller if the first nonmatching element is less.

Return Values

-1	The OID is less than the prefix.
0	The OID is in prefix.
1	The OID is greater than the prefix.

Example

```
#include <esnmp.h>
OID *q;
OBJECT *object;
if (cmp_oid_prefix(q, &object->oid) == 0)
    printf("matches prefix\n");
```

clone_oid

clone_oid — Makes a copy of the OID. This routine does not allocate an OID structure.

Syntax

```
oid clone_oid ( oid *new, oid *oid );
```

Arguments

new

A pointer to the OID structure that is to receive the copy.

oid

A pointer to the OID structure where the data is to be obtained.

Description

This routine dynamically allocates the buffer and inserts its pointer into the OID structure received. The caller must explicitly free this buffer.

Point to the OID structure that is to receive the new OID values and call this routine. Any previous value in the new OID structure is freed (using the `free_oid` routine) and the new values are dynamically allocated and inserted. To preserve an existing OID structure, initialize the new OID structure with zeros.

If the old OID structure is null or contains a null pointer to its element buffer, a new OID of [0.0] is generated.

Return Values

Null	An error or the pointer to the OID is returned.

Example

```
#include <esnmp.h>
OID oid1;
OID oid2;
:
: assume oid1 gets assigned a value
:
memset(&oid2, 0, sizeof(OID));
if (clone_oid(&oid2, &oid1) == NULL)
    DPRINTF((WARNING, "It did not work\n"));
```

free_oid

free_oid — Frees the OID structure's buffer. This routine does not deallocate the OID structure itself; it deallocates the elements buffer attached to the structure.

Syntax

```
void free_oid ( oid *oid );
```

Description

This routine frees the buffer pointed to by the *OID->elements* field and zeros the field and the NELEM structure.

Example

```
include <esnmp.h>
OID oid;
:
: assume oid was assigned a value (perhaps with clone_oid()
: and we are now finished with it.
:
free_oid(&oid);
```

clone_buf

clone_buf — Duplicates a buffer in a dynamically allocated space.

Syntax

```
char clone_buf ( char *str, int *len );
```

Arguments

str

A pointer to the buffer to be duplicated.

len

The number of bytes to be copied.

Description

One extra byte is always allocated at the end and is filled with zeros. If the length is less than zero, the duplicate buffer length is set to zero. A buffer pointer is always returned, unless there is a `malloc` error.

Return Values

Null	A `malloc` error. Otherwise, the pointer to the allocated buffer that contains a copy of the original buffer is returned.

Example

```
#include <esnmp.h>
char *str = "something nice";
char *copy;
copy = clone_buf(str, strlen(str));
```

mem2oct

mem2oct — Converts a string (a buffer and length) to an `oct` structure with the new buffer's address and length.

Syntax

```
oct *mem2oct ( oct *new, char *buffer, int *len );
```

Argument

new

A pointer to the `oct` structure receiving the data.

buffer

Pointer to the buffer to be converted.

len

Length of buffer to be converted.

Description

The `mem2oct` routine dynamically allocates the buffer and inserts its pointer into the `oct` structure. The caller must explicitly free this buffer.

This routine does not allocate an `oct` structure and does not free data previously pointed to in the `oct` structure before making the assignment.

Return Values

Null	An error occurred. Otherwise, the pointer to the `oct` structure (the first argument) is returned.

Example

```
#include <esnmp.h>
char buffer;
int len;
OCT abc;

...buffer and len are initialized to something...

memset(&abc, 0, sizeof(OCT));
if (mem2oct(&abc, buffer, len) == NULL)
    DPRINTF((WARNING, "It did not work...\n"));
```

cmp_oct

cmp_oct — Compares two octet strings.

Syntax

```
int cmp_oct ( oct *oct1, oct *oct2 );
```

Arguments

oct1

Pointer to the first octet string.

oct2

Pointer to the second octet string.

Description

The two octet strings are compared byte-by-byte to determine the length of the shortest octet string. If all bytes are equal, the lengths are compared. An octet with a null pointer is considered the same as a zero-length octet.

Return Values

-1	The string pointed to by the first `oct` is less than the second.
0	The string pointed to by the first `oct` is equal to the second.
1	The string pointed to by the first `oct` is greater than the second.

Example

```
#include <esnmp.h>
OCT abc, efg;

...abc and efg are initialized to something...

if (cmp_oct(&abc, &efg) > 0)
    DPRINTF((WARNING, "octet abc is larger than efg...\n"));
```

clone_oct

clone_oct — Makes a copy of the data in an `oct` structure. This routine does not allocate an `oct` structure; it allocates the buffer pointed to by the `oct` structure.

Syntax

```
oct clone_oct ( oct *new, oct *old );
```

Arguments

new

A pointer to the `oct` structure receiving the data.

old

A pointer to the `oct` structure where the data is to be obtained.

Description

The `clone_oct` routine dynamically allocates the buffer, copies the data, and updates the `oct` structure with the buffer's address and length. The caller must free this buffer.

The previous value of the buffer on the new `oct` structure is freed prior to the new buffer being allocated. If you do not want the old value freed, initialize the new `oct` structure before cloning.

Return Values

Null	An error occurred. Otherwise, the pointer to the `oct` structure (the first argument) is returned.

Example

```
#include <esnmp.h>
OCT octet1;
OCT octet2;
:
: assume octet1 gets assigned a value
:
memset(&octet2, 0, sizeof(OCT));
if (clone_oct(&octet2, &octet1) == NULL)
    DPRINTF((WARNING, "It did not work\n"));
```

free_oct

free_oct — Frees the buffer attached to an `oct` structure. This routine does not deallocate the `oct` structure; it deallocates the buffer to which the `oct` structure points.

Syntax

```
void free_oct ( oct *oct );
```

Description

This routine frees the dynamically allocated buffer to which the `oct` structure points, and zeros the pointer and length on the `oct` structure. If the `oct` structure is already null, this routine does nothing.

If the buffer attached to the `oct` structure is already null, this routine sets the length field of the `oct` structure to zero.

Example

```
#include <esnmp.h>
OCT octet;
:
: assume octet was assigned a value (perhaps with mem2oct()
: and we are now finished with it.
:
free_oct(&octet);
```

free_varbind_data

free_varbind_data — Frees the dynamically allocated fields in the VARBIND structure. However, this routine does not deallocate the VARBIND structure itself; it deallocates the name and data buffers to which the VARBIND structure points.

Syntax

```
void free_varbind_data ( varbind *vb );
```

Description

This routine performs a `free_oid` (vb->name) operation. If indicated by the vb->type field, it then frees the vb->value data using either the `free_oct` or the `free_oid` routine.

Example

```
#include <esnmp.h>
VARBIND *vb;

vb = (VARBIND*)malloc(sizeof(VARBIND));
clone_oid(&vb->name, oid);
clone_oct(&vb->value.oct, data);
:
: some processing that uses vb occurs here
:
free_varbind_data(vb);
```

```
free(vb);
```

set_debug_level

set_debug_level — Sets the logging level, which dictates what log messages are generated. The program or module calls the routine during program initialization in response to run-time options.

Syntax

```
void set_debug_level ( int stat, unsigned integer null );
```

Arguments

stat

The logging level. The following values can be set individually or in combination:

Level	Meaning
ERROR	Used when a bad error occurred; requires a restart.
WARNING	Used when a packet cannot be handled; also implies ERROR. This is the default.
TRACE	Used when tracing all packets; also implies ERROR and WARNING.

null

This parameter is not used by OpenVMS. It is supplied for compatibility with UNIX.

Description

The logging level will be ERROR, WARNING, or TRACE.

If you specify TRACE, all three types of errors are generated. If you specify ERROR, only error messages are generated. If you specify WARNING, both error and warning messages are generated.

To specify logging levels for the messages in your subagent, use the ESNMP_LOG routine.

Example

```
#include <esnmp.h>

if (strcmp("-t", argv[1] {
    set_debug_level(TRACE, NULL);
} else {
    set_debug_level(WARNING, NULL);
}
```

is_debug_level

is_debug_level — Tests the logging level to see whether the specified logging level is set.

Additional Information

You can test the logging levels as follows:

Level	Meaning
ERROR	Used when a bad error occurs, requiring restart.
WARNING	Used when a packet cannot be handled; this also implies ERROR.
TRACE	Used when tracing all packets; this also implies ERROR and WARNING.

Syntax

```
int is_debug_level ( int type );
```

Return Values

TRUE	The requested level is set and the ESNMP_LOG will generate output, or output will go to the specified destination.
FALSE	The logging level is not set.

Example

```
#include <esnmp.h>

if (is_debug_level(TRACE))
    dump_packet()
```

ESNMP_LOG

ESNMP_LOG — This is an error declaration C macro defined in the ESNMP.H header file. It gathers the information that it can obtain and sends it to the log.

Syntax

```
ESNMP_LOG ( level, format, ... );
```

Description

The `esnmp_log` routine is called using the ESNMP_LOG macro, which uses the helper routine `esnmp_logs` to format part of the text. Do not use these functions without the ESNMP_LOG macro. For example:

```
#define ESNMP_LOG(level, x) if (is_debug_level(level)) { \
        esnmp_log(level, esnmp_logs x, __LINE__, __FILE__);}
```

Where:

- *x* is a text string; for example, a `printf` statement.

- *level* is one of the following:

ERROR	Declares an error condition.
WARNING	Declares a warning.
TRACE	Puts a message in the log file if trace is active.

For more information about configuration options for logging and tracing, refer to the *VSI TCP/IP Services for OpenVMS Management* guide.

Example

```
#include <esnmp.h>
ESNMP_LOG( ERROR, ("Cannot open file %s\n", file));
```

__print_varbind

__print_varbind — Displays the VARBIND and its contents. This routine is used for debugging purposes. To use this routine, you must set the debug level to TRACE. Output is sent to the specified file.

Syntax

```
__print_varbind ( VARBIND *vb, int indent );
```

Arguments

vb

The pointer to the VARBIND structure to be displayed. If the *vb* pointer is NULL, no output is generated.

indent

The number of bytes of white space to place before each line of output.

set_select_limit

set_select_limit — Sets the eSNMP select error limit. For more information, see Section 6.1.

Syntax

```
set_select_limit ( char *progname );
```

Arguments

progname

The subagent name. This argument is valid with DPI versions only. With AgentX, the argument is NULL because subagents do not get names.

Return Values

ESNMP_MTHD_noError	No error was generated.

ESNMP_MTHD_genErr	An error was generated.

__set_progname

__set_progname — Specifies the program name that will be displayed in log messages. This routine should be called from the main during program initialization. It needs to be called only once.

Syntax

```
__set_progname ( char *prog );
```

Arguments

prog

The program name as taken from `argv[0]`, or some other identification for entity-calling logging routines.

Example

```
#include "esnmp.h"
__set_progname(argv[0]);
```

__restore_progname

__restore_progname — Restores the program name from the second application of the set. This routine should be called only after the `__set_progname` routine has been called. You can use this to restore the most recent program name only.

Syntax

```
__restore_progname ( );
```

Example

```
#include "esnmp.h"
__restore_progname();
```

__parse_progname

__parse_progname — Parses the full file specification to extract only the file name and file extension.

Syntax

```
__parse_progname ( file-specification );
```

Arguments

file-specification

The full file specification for the subagent.

Example

```
#include "esnmp.h"
static char Progname[100];
sprintf (Progname, "%s%.8X", __parse_progname(prog), getpid());
```

esnmp_cleanup

esnmp_cleanup — Closes open sockets that are used by the subagent for communicating with the master agent.

Syntax

```
esnmp_cleanup ( );
```

Example

```
#include "esnmp.h"
int rc = ESNMP_LIB_OK;
rc = esnmp_cleanup();
```

Return Values

ESNMP_LIB_NOTOK	There was no socket.
ESNMP_LIB_OK	Success.

Chapter 6. Troubleshooting eSNMP Problems

The eSNMP modules provided with TCP/IP Services include troubleshooting features that are useful in controlling the way your subagent works.

This chapter describes:

- How to modify the subagent error limit (Section 6.1)

- How to modify the default subagent timeout value (Section 6.2)

- Log files (Section 6.3)

For additional information about troubleshooting SNMP problems, see the *VSI TCP/IP Services for OpenVMS Management* guide.

6.1. Modifying the Subagent Error Limit

In certain circumstances, some subagent programs might enter a loop where a `select()` call repeatedly returns a -1 error value. (Note that standard SNMP subagents and the Chess example provided in TCPIP$EXAMPLES should not exhibit this behavior.)

You can define the logical name TCPIP$SNMP_SELECT_ERROR_LIMIT to modify the number of times a -1 error value can be returned from a `select()` call.

The valid TCPIP$SNMP_SELECT_ERROR_LIMIT values range from 1 to less than $2^{32} - 1$ (default 100). When defining the error limit, remember:

- Do not use commas when defining the number.

- If you defined the limit as 0, no limit is set.

- A defined value greater than or equal to 4000000000 triggers warning messages.

For example, to define TCPIP$SNMP_SELECT_ERROR_LIMIT to limit the number of times a -1 error value is returned to 1, 000, enter the following command:

```
$ DEFINE/SYSTEM TCPIP$SNMP_SELECT_ERROR_LIMIT 1000
```

6.2. Modifying the Subagent Timeout

You can define the logical name TCPIP$ESNMP_DEFAULT_TIMEOUT to modify the default time allowed (3 seconds) before timeout occurs because of the lack of response by the subagent to the master agent. The ability to define the timeout is especially useful for slower systems and systems with heavy network traffic. The logical name is translated at startup time.

The TCPIP$ESNMP_DEFAULT_TIMEOUT value is from 0 to 60 seconds. (You should use 0 only for testing purposes, such as simulating problems on a heavily loaded host or network.) If the value you specify contains non-numeric digits or is outside the allowed range, the default value of 3 seconds is used.

For example, to define TCPIP$ESNMP_DEFAULT_TIMEOUT to time out after 6 seconds of inactivity between the master agent and subagents, enter the following command:

```
$ DEFINE/SYSTEM TCPIP$ESNMP_DEFAULT_TIMEOUT 6
```

When a subagent registers with the master agent, it can specify a value that overrides the value you set with logical name TCPIP$ESNMP_DEFAULT_TIMEOUT. The standard MIB II and Host Resources MIB subagents use the default value of 3 seconds. Refer to the description of the `esnmp_register` routine for more information.

6.3. Log Files

All output redirected from SYS$OUTPUT for the SNMP agent process is logged to *.LOG files in the SYS$SYSDEVICE:[TCPIP$SNMP] directory. Output redirected from SYS$ERROR is logged to *.ERR files in the same directory.

Output redirected from SYS$OUTPUT for the agent process is logged to the following files:

- TCPIP$ESNMP.LOG (for the master agent)

- TCPIP$OS_MIBS.LOG (for the MIB II)

- TCPIP$HR_MIB.LOG (for the Host Resources MIB)

Output redirected from SYS$ERROR is logged to the following files:

- TCPIP$ESNMP.ERR (for the master agent)

- TCPIP$OS_MIBS.ERR (for the MIB II)

- TCPIP$HR_MIB.ERR (for the Host Resources MIB)

Data is flushed to the log files when the corresponding process terminates. Each invocation of the TCPIP$SNMP_RUN.COM procedure purges these files, retaining at least the last seven versions (the exact number depends on the value of the CLUSTER_NODES system parameter).

The log files are located in the SYS$SYSDEVICE:[TCPIP$SNMP] directory along with the TCPIP $SNMP_CONF.DAT file, which is a text representation of the SNMP configuration data generated by the master agent during startup.

The contents of the SNMP log files are written to SYS$SYSDEVICE:[TCPIP$SNMP] when the process stops or when you stop it (for example, by entering the STOP/ID= *xxx* command). After a process restarts, it creates a new version of the files. If a process executes without errors, *.ERR files might not be created.

Writing to SYS$OUTPUT and SYS$ERROR from custom subagents is controlled by qualifiers on the RUN command in the TCPIP$EXTENSION_MIB_RUN.COM procedure. See Chapter 3 for information about including extension subagent commands in the startup procedure.

Custom subagents that do not write to SYS$OUTPUT and SYS$ERROR might not produce a .LOG or .ERR file.

TCP/IP Services does not support writing log files to locations other than the SYS$SYSDEVICE: [TCPIP$SNMP] directory.

The log files contain startup and event information and additional messages, depending on the logging level specified for an agent. The SNMP logging facility uses three logging levels:

- TRACE (logs trace, warning, and error messages)

- WARNING (logs warning and error messages)

- ERROR

For the master agent and standard subagents, the logging level is WARNING. Log files for these processes include messages for WARNING and ERROR events. The chess example does not have a default log level. Therefore, no log messages appear. To specify a default log level for custom subagents, you can use the standard API call `set_debug_level` (see Chapter 5 for more information). Because the chess example subagent does not use a default, messages are captured only if you specify tracing. For information about getting trace logs, refer to the *VSI TCP/IP Services for OpenVMS Management* guide.