



# Treck SNMP User Manual



**Complete Internet Solutions**

**Technical Support:**

5041 Lamart Drive #240 Riverside, California 92507  
Phone: (909) 787-7056 Fax: (909) 787-8803

**Corporate Headquarters:**

Treck, Inc. 431 Ohio Pike Suite 210 North Cincinnati, Ohio 45255  
Phone: (513) 528-5732 Fax: (513) 528-5740

# Contents

<b>1 Introduction .....</b>	<b>7</b>
1.1 SNMP Overview .....	7
1.2 MIB Structure .....	8
1.3 SNMPv1 Messages .....	9
1.3.1 GetRequest PDU .....	10
1.3.2 GetNextRequest PDU .....	10
1.3.3 SetRequest PDU .....	10
1.3.4 GetResponse PDU .....	10
1.3.5 Trap PDU .....	10
1.4 SNMPv2c Messages .....	11
1.5 SNMP Security .....	11
1.6 SNMP Messages Language and Encoding Rules .....	13
1.7 SNMP Messages and PDU Formats .....	15
1.7.1 SNMPv1 Messages and PDU Formats .....	15
1.7.2 SNMPv2c Messages and PDU Formats .....	15
1.7.3 SNMPv3 Messages and PDU Formats .....	16
1.7.4 Recommended Literature .....	16
<b>2 Treck SNMP Agent Design .....</b>	<b>17</b>
<b>3 Treck SNMP Agent Description .....</b>	<b>19</b>
3.1 How To Build the Agent .....	19
3.1.1 Building the SNMP Agent Library (MS-DOS only) .....	20
3.2 Running the Agent .....	20
3.2.1 tfSnmpdMain .....	20
3.2.2 tfSnmpdStop .....	20
3.3 Macros .....	21
3.4. SNMPv3 initialization .....	26
3.4.1. Overview .....	26
3.4.2. usmUserTable .....	27
3.4.3. vacmContextTable .....	29
3.4.4. vacmSecurityToGroupTable .....	30
3.4.5. vacmAccessTable .....	31
3.4.6. vacmViewTreeFamilyTable .....	32
3.4.7. snmpCommunityTable .....	33
3.4.8. snmpNotifyTable .....	34
3.4.9. snmpNotifyFilterTable .....	35
3.4.10. snmpTargetAddrTable .....	36
3.4.11. snmpNotifyFilterProfileTable .....	37
3.4.12. snmpTargetParamsTable .....	38
3.4.13. Default Configuration .....	39
3.5 Running treck SNMP without Treck TCP/IP .....	41
3.5.1. system settings in trdefs.h: .....	41
3.5.2. System-dependent functions required by SNMP: .....	42
3.6 Testing .....	43
3.7 MIB Compiler and Code Generator Procedures .....	44
3.7.1 MIB Compiler .....	44
3.7.2 Installing the mib compiler: .....	44
3.7.3 Generating the code with the mib compiler .....	44
3.7.4 Hooking the generated code in the SNMP agent: .....	45
3.8 Configuration APIs .....	46
3.9 Protocol Stack-Specific functions .....	46
3.10 Contents of the Distribution media .....	47
3.10.1 Source Files .....	47
3.10.2 Header Files .....	47

<b>4 Programmer's Reference .....</b>	<b>48</b>
4.1 Core agent functions .....	49
4.1.1 tfSnmpdMain .....	49
4.1.2 tfSnmpdStop .....	50
4.2 Non-Runtime Initialization functions .....	51
4.2.1 tfModifyAuthPassword .....	51
4.2.2 tfModifyPrivacyPassword .....	52
4.2.3 tfSetSnmpPort .....	53
4.2.4 tfModifySysDescr .....	54
4.2.5 tfModifySysObjid .....	55
4.2.6 tfModifySysServices .....	56
4.2.7 tfSetColdStartTrap .....	57
4.3 Configuration API functions .....	58
4.3.1 tfAddCommunity .....	58
4.3.2 tfAddTrapEntry .....	59
4.3.3 tfDeleteAccessEntry .....	60
4.3.4 tfAddCommunityTableEntry .....	61
4.3.5 tfDeleteCommunity .....	63
4.3.6 tfDeleteCommunityTableEntry .....	64
4.3.7 tfDeleteContextEntry .....	65
4.3.8 tfDeleteSec2GroupEntry .....	66
4.3.9 tfDeleteTrapEntry .....	67
4.3.10 tfDeleteUsmEntry .....	68
4.3.11 tfDeleteVtfEntry .....	70
4.3.12 tfDisplayAccessEntry .....	71
4.3.13 tfDisplayContextEntry .....	72
4.3.14 tfDisplaySec2GroupEntry .....	73
4.3.15 tfDisplaySysContact .....	74
4.3.16 tfDisplaySysDescr .....	75
4.3.17 tfDisplaySysObjectId .....	76
4.3.18 tfDisplaySysObjectId .....	77
4.3.19 tfDisplaySysLocation .....	78
4.3.20 tfDisplaySysName .....	79
4.3.21 tfDisplayTrapEntry .....	80
4.3.22 tfDisplayUsmEntry .....	81
4.3.23 tfDisplayVtfEntry .....	82
4.3.24 tfInsertAccessEntry .....	83
4.3.25 tfInsertContextEntry .....	85
4.3.26 tfInsertSec2GroupEntry .....	86
4.3.27 tfInsertUsmEntry .....	87
4.3.28 tfInsertVtfEntry .....	89
4.3.29 tfModifyCommunity .....	90
4.3.30 tfModifySysContact .....	91
4.3.31 tfModifySysLocation .....	92
4.3.32 tfModifySysName .....	93
4.3.33 tfModifyTrapEntry .....	94
4.3.34 tfNgAddTrapEntry .....	95
4.3.35 tfNgDeleteTrapEntry .....	97
4.3.36 tfNgDisplayTrapEntry .....	98
4.3.37 tfNgModifyTrapEntry .....	99
4.4 Protocol stack-specific functions .....	100
4.4.1 tfGetNumberIfs .....	100
4.4.2 tfIcmpGroupInfoGet .....	101
4.4.3 tfIfGroupInfoGet .....	103
4.4.4 tfIpAddrTableEntryGet .....	105
4.4.5 tfIpGroupInfoGet .....	107
4.4.6 tfIpRouteTableEntryGet .....	109

---

4.4.7 tfN2mTableEntryGet .....	110
4.4.8 tfTcpGroupInfoGet .....	112
4.4.9 tfTcpTableEntryGet .....	114
4.4.10 tfUdpGroupInfoGet .....	116
4.4.11 tfUdpTableEntryGet .....	117
4.4.12 tfWriteAdminStat .....	119
4.4.13 tfWriteIp .....	120
4.4.14 tfWriteIpRouteEntry .....	121
4.4.15 tfWriteN2mEntry .....	122
4.4.16 tfWriteTcpState .....	124



# 1 Introduction

## 1.1 SNMP Overview

The Simple Network Management Protocol (SNMP) is today the de-facto industry standard to manage devices on data communication networks, telecommunication systems and other globally reachable devices. Practically every organization dealing with computers and related devices expects to be able to monitor, diagnose and configure each such device from a central Network Operations Center (NOC). In order to do so, the applications at the NOC need to actively interact with the managed heterogeneous devices spread all over the organization and beyond, often across continents. SNMP is the protocol that enables these interactions.

The SNMP architectural model consists of the following key elements:

- Management station
- Management agent
- Management information base (MIB)
- Network management protocol

The management station is typically a stand-alone device, but it may be a capability implemented on a shared system. In either case, the management station serves as the interface for the human network manager into the network management system. At a minimum, the management station will have:

A set of management applications for the data analysis, fault recovery, etc.

A user interface by which the network manager may monitor and control the network.

The capability of translating the network manager's requirements into the actual monitoring and control of remote elements of the network.

A database of information extracted from the MIBs of all the managed entities in the network.

Only the last two elements are the subject of SNMP standardization.

The other active element in the network management system is the managed agent. Key platforms, such as hosts, routers or switches, may be equipped with SNMP agents so that they can be managed from a management station. The managed agent responds to requests for information and actions from the management station and may asynchronously provide the management station with important but unsolicited information.

Resources in the network may be managed by representing these resources as objects. Each object is a data variable representing one aspect of the managed agent. The collection of objects is referred to as MIB. The MIB functions as a collection of access points at the agent for the management station. A management station performs the monitoring function by retrieving the value of MIB objects. A management station can cause an action to take place at an agent or can change the configuration settings at an agent by modifying the value of specific variables.

The management station and agents are linked by the SNMP protocol, which includes the following capabilities:

- Enabling the management station to retrieve the value of objects at the agent
- Enabling the management station to modify the value of objects at the agent
- Enabling the agent to notify the management station of significant events.

The first SNMP version (SNMPv1) was introduced in the late 1980s and is a de-facto standard for multi-vendor network management. SNMPv2c (1992) provided more functionality and greater efficiency than SNMPv1 by expanding on error identifications and by introducing the GETBULK operator. It also introduced a simplified TRAP format. The functionality of SNMPv3 (finalized 1999) is defined in RFCs 2570-2575.

## 1.2 MIB Structure

MIBs are either standard or proprietary. Standard MIBs are published by the Internet Engineering Task Force (IETF) as Requests For Comments (RFCs) (see <http://www.rfc-editor.org/rfc.html>). Proprietary MIBs can be designed and implemented by any organization as long as the MIB is defined under an enterprise specific object ID (see below). Of all standard or proposed standard MIBs, the group of managed objects known as MIB-II is the most basic and the most popular MIB. All SNMP agents implement a subset of MIB-II (some MIB-II components are optional for most systems and some components were deprecated by later RFCs). MIB-II is defined in RFC 1213.

MIB is a rooted tree structure that contains globally unique object identifiers (OIDs). To describe an object, you walk from the root of the tree through the branches to the object, concatenating ISO numbers for OID or text strings for Object Descriptors. While walking through the tree, subordinate limbs may be sparse. This allows for private MIBs to be attached to the standard MIB in any fashion. From an unlabeled root, the limbs in the internet tree are shown below:

A specific instance of a MIB element is defined by its position in the rooted tree, followed by a string specifying its instance.

### Example 1

Part of the MIB-II is a count of how long a device has been up. The Object ID for this element is:

---

```
iso.org.dod.internet.mgmt.mib.system.sysUpTime  
1.3.6.1.2.1.1.3
```

---

In addition to knowing its Object ID, the instance must be specified. There is only one system group per “mib view”, so a zero is appended to the Object ID to specify this instance of sysUpTime. The objects with this property are called scalar objects, or scalars.

---

```
iso.org.dod.internet.mgmt.mib.system.sysUpTime.0  
1.3.6.1.2.1.1.3.0
```

---

### Example 2

Another part of the MIB-II is a count of how many octets a device has received. The Object ID for this element is:

---

```
iso.org.dod.internet.mgmt.mib.interfaces.ifTable.ifEntry.ifInOctets  
1.3.6.1.2.1.2.2.1.10
```

---

In addition to knowing its Object ID, the instance must be specified. Since devices may have more than one interface, the interface number is appended to the Object ID. The objects with this property are called tabular objects

---

```
iso.org.dod.internet.mgmt.mib.interfaces.ifTable.ifEntry.ifInOctets.interface1  
1.3.6.1.2.1.2.2.1.10.1
```

---

The instance does not have to be indexed by an integer. Various MIBs define tables with octet strings, IP addresses, and OIDs as indices. The index does not have to be simple and may consist of a number of fields. For example, the TCP Connections Table of MIB-II is indexed by local address (type IP address), local port (type integer), remote address (type IP address), remote port (type integer) – in that order.

The enterprises node is defined as OID `iso.org.dod.internet.private.enterprises(1.3.6.1.4.1)`. All proprietary MIBs are attached to this node via Private Assigned Enterprise Numbers which are issued by IETF following the request from an organization. For example, all Cisco MIBs contain the enterprise number 9, and Cisco path is `1.3.6.1.4.1.9`, Oracle is under enterprise number 111, etc. Close to 10,000 enterprise numbers are issued to date. The private enterprise numbers are listed in RFC 1700.

### 1.3 SNMPv1 Messages

All SNMP messages contain a Protocol Data Unit (PDU) as a part of the message. Each PDU includes variable bindings field(s) by which the requested object is defined. Each variable bindings (varbind) field consists of two entities: the identity (basically OID) and the value. The number of varbinds in a PDU is not limited.

The SNMPv1 defines five type of PDUs. These PDUs are:

- GetRequest PDU
- GetNextRequest PDU
- SetRequest PDU
- GetResponse PDU
- Trap PDU

Of these, the first three may be issued only by a manager, and the last two only by an agent.

### 1.3.1 GetRequest PDU

The manager issues this PDU when it knows the exact OID of the variable it is going to retrieve. For example, in the example above, the manager may issue this PDU to get sysUpTime with the following OID: 1.3.6.1.2.1.1.3.0. If, on the other hand, the manager asks for 1.3.6.1.2.1.1.3, the agent would respond with an error since it won't be able to find a variable with the indicated OID. The value field in the varbinds is set to zero in GetRequest PDU.

### 1.3.2 GetNextRequest PDU

This PDU is almost identical to the GetRequest PDU. The only difference is the following: In the GetRequest PDU, each variable in the varbinds field refers to an object instance whose value is to be returned. In the GetNextRequest PDU, for each variable, the agent is to return the value of the object instance that is next in lexicographical order. This difference, though, has tremendous implications. It allows the manager to discover the structure of a MIB dynamically. It also provides an efficient mechanism for searching a table whose entries are unknown. Suppose the manager does not know anything about the MIBs the agent implements. The manager might start with issuing GetNextRequest with the varbind identity field(OID) .1. Since any path in the MIB tree must contain the root, this request is guaranteed to succeed. Suppose further, the agent responds with the lexicographically next object 1.3.6.1.2.1.1.1.0 (sysDescr) and the corresponding value field. The manager then issues a new GetNextRequest with the OID 1.3.6.1.2.1.1.1.0. If the agent implements all object in the system group of MIB-II (as it should), it responds with 1.3.6.1.2.1.1.2.0 (sysObjectID). The manager issues another GetNextRequest with the OID 1.3.6.1.2.1.1.2.0 and so on. This strategy allows the manager to retrieve the entire MIB, and the end is signaled by the agent when it issues the error code noSuchName (see error code definitions below). The value field in the varbinds is set to zero in GetNextRequest PDU.

### 1.3.3 SetRequest PDU

This PDU is used by a manager to write an object value. Thus the varbinds list in the SetRequest PDU include both OID and a value to be assigned to each OID listed. As in GetRequest PDU, the manager must know the exact OID of the variable it is going to set.

### 1.3.4 GetResponse PDU

This PDU is issued by the agent in response to GetRequest, GetNextRequest and SetRequest PDUs. In the response to the first two, the agent simply populates the value field of the varbinds, and in case of the SetRequest, the varbinds list is returned unchanged.

### 1.3.5 Trap PDU

This PDU is issued by the SNMP agent to provide the manager with an asynchronous notification of some significant event. The manager is responsible for catching and interpreting a Trap PDU. The traps may be generic or enterprise-specific. The following generic traps are defined:

- coldStart (value 0). The agent must send this trap on initialization
- warmStart (1). The agent sends this trap when it is initialized in such a way that its configuration and protocol implementation is not altered.
- linkDown (2). Failure of one of the communication links.
- linkUp (3). One of the communication links has come up.
- authenticationFailure (4). The agent received a PDU that failed authentication.
- egpNeighborLoss (5). Only for devices implementing EGP protocol.
- enterpriseSpecific (6). Some specific event defined in the enterprise MIB has occurred.

Each trap PDU contains generic-trap and specific-trap fields. For generic traps 0 – 5 above, the specific-trap field is set to zero, and for an enterprise-specific trap this field is set to the value defined in the enterprise MIB.

## 1.4 SNMPv2c Messages

In addition to the messages defined in SNMPv1, SNMPv2 defines GetBulkRequest. Its purpose is to minimize the number of protocol exchanges required to retrieve a large amount of management information. The GetBulkRequest uses the same selection principle as the GetNextRequest, i.e. selection is always of the next object instance in lexicographical order. The difference is that, with GetBulkRequest it is possible to specify that multiple lexicographical successors be selected. It is achieved through the use of two field in the PDU: non-repeaters and max-repetitions. The non-repeaters field specifies the number of variables in the varbinds list for which a single lexicographical successor is to be returned. The max-repetitions field specifies the number of lexicographical successors to be returned for the remaining variables in the varbinds list.

SNMPv2 specification also includes a Report PDU, but without a definition of how and when to use it. This agent utilizes the Report PDU to convey SNMPv3 security violations (see below).

## 1.5 SNMP Security

SNMPv1 and SNMPv2c provide only very limited security protection in form of a community field which is a part of every SNMP message. The community field is a printable octet string, usually with the length not exceeding some predefined maximum, and functions similar to a password. Most of the agents implement different communities for retrieval and modification operations, though this is not explicitly required by the protocol. The agent may choose to accept more than one community for each operation. This is implementation-dependent. The manager(s) and the agent(s) have to agree by means outside of the SNMP protocol on which communities will be accepted to authenticate the message. Upon receiving the SNMP message, the agent compares the community field in the incoming message with those to which it was configured, and, if a match is found, considers the message properly authenticated and continues with processing. If no match is found, the agent stops processing, and does not issue the GetResponse PDU. The manager may request to be notified of such events by setting snmpEnableAuthenTraps object of MIB-II to 'enabled'. In this case, when authentication failure occurs, the agent issues an authenticationFailure trap to the manager IP address(es) for which it was configured.

The SNMPv3 protocol was introduced to correct the weak security protection of SNMPv1 and SNMPv2c. Specifically SNMPv3 protects against the following principal threats:

- **Modification of information.** An entry could alter an in-transit message generated by an authorized entry in such a way as to effect unauthorized management operations, including the setting of object values. The essence of this threat is that an authorized entity could change any management parameter, including those related to configuration, operations, and accounting.
- **Masquerade.** Management operations that are not authorized for some entity may be attempted by that entity by assuming the identity of an authorized entity.
- **Message stream modification.** SNMP is designed to operate over a connectionless transport protocol. There is a threat that SNMP messages could be reordered, delayed, or duplicated to effect unauthorized management operations.
- **Disclosure.** An entry could observe exchanges between a manager and an agent and thereby learn the values of managed objects and learn of notifiable events.

SNMPv3 is not intended to secure against the following threats:

- Denial of service. An attacker can prevent exchanges between manager and agent.
- Traffic analysis. An attacker can observe the general pattern of traffic between managers and agents.

The protection that SNMPv3 provides against the four threats listed above is achieved through the use of the User-Based Security Model and the View-Based Access Control Model.

The User-Based Security Model (USM) is defined in RFC2574, and consists of:

- Authentication. Provides data integrity and data origin authentication. The message authentication code, HMAC, with either the hash function MD5 or SHA-1, provides authentication.
- Timeliness. Protects against message delay or replay.
- Privacy. Protects against disclosure of message payload. The cipher block chaining mode (CBC) of DES is used for encryption.
- Message format. Defines format of msgSecurityParameters field, which supports the functions of authentication, timeliness and privacy, and contains the following elements:
  1. msgAuthoritativeEngineID. A unique identifier of an SNMP entity.
  2. msgAuthoritativeEngineBoots. Represents the number of times that this SNMP engine has initialized itself since its initial configuration.
  3. msgAuthoritativeEngineTime. Represents the number of seconds since this authoritative SNMP engine last incremented the msgAuthoritativeEngineBoots object.
  4. msgUserName. The principal on whose behalf the message is being exchanged.
  5. msgAuthenticationParameters. HMAC message authentication code, or null if authentication is not used.
  6. msgPrivacyParameters. A value used to form the initialization vector in the DES CBC algorithm.
- Key management. Defines procedures for key generation, update and use.

In addition, each SNMP entity maintains a MIB, usmUser group, which consists of a single scalar object, usmUserSpinLock, and also the table usmUserTable which contains information about authorized users.

The View-Based Access Control Model (VACM) is defined in RFC2575 and contains the following elements:

- Groups. Defines a set of zero or more <securityModel, securityName> tuples on whose behalf SNMP management objects can be accessed.
- Security Level. The access rights for groups.
- Contexts. A subset of the object instances in the local MIB, providing a way of aggregating objects into collections with different access policies.
- MIB views. Defines the set of rules to restrict the access of particular groups to a subset of the managed objects.
- Access Policy. Enforces sets of access privileges for different users.

Each SNMP entity maintains a VACM MIB which consists of the following tables:

- vacmContextTable
- vacmSecurityToGroupTable
- vacmAccessTable
- vacmViewTreeFamilyTable and vacmViewSpinLock scalar object.

## 1.6 SNMP Messages Language and Encoding Rules

SNMP operations occur through message exchange over a message transport service. That calls for a unified message format. Out of existing formal languages (XDR is one example), the SNMP protocol adopted a subset of the Abstract Syntax Notation One (ASN.1) language (standardized by CCITT, X.208, and ISO, ISO 8824). This format, called an abstract syntax, is independent of the representation of data on any particular computer system. Three kinds of managed objects are defined in ASN.1:

1. Types. Define the new data structures. ASN.1 types include: BOOLEAN(1), INTEGER(2), BIT STRING(3), OCTET STRING(4), NULL(5), SEQUENCE OF(16), OBJECT IDENTIFIER(6), etc.
2. Values. Instances of a type.
3. Macros. Used to change the actual grammar of ASN.1

ASN.1 has only one “operation”, the assignment statement which uses the “:=” operator. All comments begin with two hyphens “—“ and end either with another set of hyphens or with the end of line character.

Example 1

```
Status ::= INTEGER { up(1), down(2), testing(3) }
myStatus Status ::= up - 1
```

To transmit a message, it must be first converted into a string of octets. A transfer syntax specifies the format of the converted data. SNMP protocol uses a subset of the Basic Encoding Rules (BER) (standardized by CCITT, X.209, and ISO, ISO 8825) to define the format of encoded (serialized) messages. The encoding is based on on the use of type-length-value structure. That is, any ASN.1 value can be encoded as a triplet with the following components:

- Tag: indicates the ASN.1 type.
- Length: indicates the length of the actual value representation
- Value: represents the value of the ASN.1 type as a string of octets.

The tag field consists of eight bits (most significant bit is 8, least significant bit is 1), with the bits 8,7 denoting class – Universal {00}, or Application {01}, or Context-specific {10}, or Private {11}; bit 6 is “F” bit indicating Primitive {0} or Constructed {1}; bits 5 to 1 denote the tag number. Some defined tag numbers are:

Type	Tag Class	Tag Number	Value of Tag Field
INTEGER/Integer32	UNIVERSAL	0x02	0x02
OCTET STRING	UNIVERSAL	0x04	0x04
NULL	UNIVERSAL	0x05	0x05
SEQUENCE	UNIVERSAL	0x10	0x30
IpAddress	APPLICATION	0x00	0x40
Gauge/Gauge32	APPLICATION	0x02	0x42
TimeTicks	APPLICATION	0x03	0x43

Note that SEQUENCE is the only constructed (versus primitive) type in this list.

Some examples of ASN.1 encoding:

1. type is INTEGER, value = 100 (decimal)  
Encoding:  
Tag (1 octet): 0x02  
Length (1 octet) 0x01  
Value (1 octet) 0x64
2. type is OCTET STRING, value = "Treck"  
Encoding:  
Tag (1 octet) 0x04  
Length (1 octet) 0x05  
Value (5 octets) 0x45, 0x6c, 0x6d, 0x69, 0x63
3. type is IPAddress, value = 192.168.100.9  
Encoding:  
Tag (1 octet) 0x40  
Length (1 octet) 0x04  
Value (4 octets) 0xc0, 0xa8, 0x64, 0x09
4. type is NULL  
Encoding:  
Tag (1 octet) 0x05  
Length (1 octet) 0x00

## 1.7 SNMP Messages and PDU Formats

### 1.7.1 SNMPv1 Messages and PDU Formats

The SNMPv1 message consists of three fields, encoded in ASN.1 format:

- Version – INTEGER with the value 0
- Community string – OCTET STRING
- PDU – SEQUENCE OF fields

Two PDU formats are defined, one for GetRequest, GetNextRequest, GetResponse and SetRequest, and another for Trap.

The Request or Response PDU consists of the following fields, encoded in ASN.1 format:

- request-id – INTEGER
- error-status – INTEGER
- error-index – INTEGER
- variable-bindings – SEQUENCE

The request-id field is filled by the issuer of GetRequest, GetNextRequest, or SetRequest (the manager). In GetResponse message this field remains identical with the corresponding request message. This field allows the manager to separate duplicate responses for the same request from responses for other requests. The error-status and error-index fields are zeroed in the request messages, and filled in by the agent in GetResponse if the execution of the request results in error. For the definitions of error-status values see RFC 1157. The error-index field points to the variable-bindings field causing the error, and is 1-based.

The variable-bindings field consists of a sequence of V1-VarBind. Each V1-VarBind consists of two fields, encoded in ASN.1 format:

- identity – OBJECT\_IDENTIFIER
- value – depends on ASN.1 type of value

The Trap PDU consists of the following fields, encoded in ASN.1 format:

- enterprise – OBJECT\_IDENTIFIER
- agent-addr – IPAddress
- generic-trap – INTEGER
- specific-trap – INTEGER
- time-stamp – TimeTicks
- variable-bindings – SEQUENCE

The enterprise field relays the agent identity. For example, if the company Generic Networks obtained the Enterprise Number 8888 from IETF, and they manufacture three type of products (bridges, routers and hubs), and the product router contains five different categories, then the agent in the category 4 of the router product might be assigned the enterprise .1.3.6.1.4.1.8888.2.4. The agent-addr field contains the IP address of the agent issuing the trap, or the address of a network entity on whose behalf the agent issued the trap.

### 1.7.2 SNMPv2c Messages and PDU Formats

The SNMPv2c message consists of three fields, encoded in ASN.1 format:

- Version – INTEGER with the value 1
- Community string – OCTET STRING
- PDU – SEQUENCE OF fields

Two PDU formats are defined, one for GetRequest, GetNextRequest, GetResponse, SetRequest, Report and Trap, and another for GetBulk.

The first type is identical with the SNMPv1 Request or Response PDU, and the second is almost identical, except that the meanings of fields error-status and error index becomes non-repeaters and max-repetitions. The error-status values have been expanded in SNMPv2. For the definitions of error-status values see RFC 1905.

### 1.7.3 SNMPv3 Messages and PDU Formats

The SNMPv3 message consists of the following fields, encoded in ASN.1 format:

- msgVersion – INTEGER with the value 3
- msgGlobalData
  - msgID – INTEGER
  - msgMaxSize – INTEGER
  - msgFlags – INTEGER
  - msgSecurityModel - INTEGER
- msgSecurityParameters
  - msgAuthoritativeEngineID – OCTET STRING
  - msgAuthoritativeEngineBoots - INTEGER
  - msgAuthoritativeEngineTime – TimeTicks
  - msgUserName - OCTET STRING
  - msgAuthenticationParameters - OCTET STRING
  - msgPrivacyParameters - OCTET STRING
- ScopedPDU (plaintext or encrypted)
  - ContextEngineID – OCTET STRING
  - ContextName – OCTET STRING
  - PDU – SEQUENCE OF fields

For the full definitions of fields in SNMPv3 message, see RFC 2572 and RFC 2574. The PDU format is identical to that of SNMPv2.]

### 1.7.4 Recommended Literature

1. “Understanding SNMP MIBs”, by David Perkins and Evan McGinnis, Prentice Hall, 1997
2. “SNMP, SNMPv2, SNMPv3, and RMON 1 and 2”, by William Stallings, 3<sup>rd</sup> edition, Addison-Wesley, 1999
3. “Abstract Syntax Notation One (ASN.1). The Tutorial and Reference”, by Douglas Steedman, Technology Appraisals Ltd., 1993
4. “The Simple Book. An Introduction to Internet Management”, by Marshall Rose, 2<sup>nd</sup> edition, Prentice Hall, 1994

## 2 Treck SNMP Agent Design

Treck SNMP Agent is an embedded trilingual (v1/v2c/v3) SNMP agent fully independent of processor, RTOS and development tools. It is an extensible agent allowing adding an unlimited number of MIBs to the code automatically by running Treck Automatic Code Generator. Treck SNMP Agent is fully compliant to the following RFCs:

- RFC 1155: SNMP and SNMP MIB
- RFC 1157: SNMP, version 1
- RFC 1212: Concise MIB definitions
- RFC 1213: MIB II for TCP/IP
- RFC 1215: Convention for defining traps for use with the SNMP
- RFC 1901: Introduction to Community-based SNMPv2
- RFC 1902: Structure of Management Information (SMI), v2
- RFC 1903: Textual Conventions
- RFC 1904: Conformance Statements
- RFC 1905: Protocol Operations
- RFC 1906: Transport Mappings
- RFC 1907: Management Information Base
- RFC 1908: Co-existence between Version 1 and Version 2
- RFC 2571: An Architecture for Describing SNMP Management Frameworks
- RFC 2572: Message Processing and Dispatching for SNMP
- RFC 2574: User-based Security Model (USM) for SNMPv3
- RFC 2575: View-based Access Control Model

The agent can be compiled as v1 only (`#define TM_SNMP_VERSION 1`), or as a bilingual v1/v2c (`#define TM_SNMP_VERSION 2`), or as a trilingual v1/v2c/v3 (`#define TM_SNMP_VERSION 3`). When compiled as v1, the agent accepts only SNMP v1 messages and issues v1 traps. When compiled as bilingual, it accepts v1 as well as v2 messages and issues responses accordingly. It sends both v1 and v2 traps to the configured trap sinks. In the trilingual mode it accepts all three version messages and issues traps in all three formats.

Treck SNMP Agent supports two architectural models of operation:

- **Non-blocking SNMP Agent:** Incoming SNMP packets are delivered to the SNMP agent in-line, which means the UDP directly calls the SNMP via a subroutine call. The SNMP parses the SNMP packet, and returns a response immediately, in a single pass through the TCP/IP stack. Note that this also forces a non-blocking implementation of the SNMP agent. This architecture is well suited to implementations in which all variables managed via SNMP are readily available, which is typical of single-board systems.
- **Multi-tasking Agent:** Requires an SNMP task to process SNMP messages. The operating system or RTOS starts a task that binds itself to the SNMP port. Once an SNMP packet is parsed by UDP stack, it is put on the receive queue of the SNMP socket. The SNMP task is put in a ready-to-run state. The SNMP task reads the SNMP packet off the receive queue, processes the request, and calls the UDP transmit routines to send the response. This decouples the SNMP processing from the operating system processing, and thus simplifies debugging.

The following features highlight the advantages of Treck SNMP Agent design.

- **Compact code size.** Care was taken to reduce the agent code size: the v1/v2c code RAM size is 48K, ROM 14.5K;

for the v1/v2c/v3 code the corresponding figures are – RAM 110K, ROM 18.6K. In both cases stack size required does not exceed 2K. These figures can be reduced if full compliance to the corresponding RFCs is not required.

- **Optimized SET request processing.** Normally the agents require four to five passes to process a SET request. Due to a unique Treck SNMP Agent design, this agent does it in only two passes. During the first pass two lists are dynamically built – one collecting all variable bindings in the SET request, together with the existing instances of these bindings, to allow a clean rollback in case an actual setting of the variable fails in the second pass. Another list collects all variable bindings with the read-create clause, and at the end of the first pass the decision is made whether the request relating to a conceptual row table operation is valid. If so, and the request is to create a non-existing row, the row is created immediately and populated during the second pass. If it is found at the end of the first pass that a violation took place, or an operation is requested on an object with other than read-create clause, no second pass becomes necessary, and the response PDU with the indicated error is built right away. Any number of different row or tables operations can be processed.
- **Optimized and unified search of table and scalar objects for GET and GETNEXT requests.** A standard function with two arguments is called each time the table or the group object is to be accessed. The first argument is the flag indicating if an exact match is requested, and the second is the pointer to the corresponding structure with the index pointing to the specific row (groups are considered tables with one row). The function returns success if the row is found (exact match), or the lexicographically next row is found (non-exact match). The function returns failure if the row is not found (exact match), or there is no next row in the table (non-exact match). The sorting based on table index of any complexity is done inline inside the function in two passes and does not involve dynamic memory allocation. In case of success the functions fills the related structure with the row object instances.
- **Separate processing of SNMP v3 messages.** At the onset the agent finds out if the request is v3, and if so, the processing (authentication, privacy, access control, timeliness and other checks) is done separately, then merged with the regular variable bindings processing, and then separated again to build a specific v3 response.
- **The Configuration API functions** enable the user to add / modify / delete the vital parameters at runtime. The example would be changing communities fields or trap sink addresses while the agent is running.
- **Treck Automatic Code Generator** based on the *libsmi* MIB Compiler which adds necessary code based on MIB files to the core agent. The SNMP stub functions to get and set MIB objects and rows creation and deletion stub functions are generated, so the developer unfamiliar with the SNMP protocol can fill in the implementation-dependent code inside these stubs. An unlimited number of MIBs can be added to the agent this way, with two source and two header files per MIB added to the code. The developer has to be concerned with only one source file containing stub functions.

## 3 Treck SNMP Agent Description

The items in this Chapter describe building of an agent, running the agent, specific macros, testing the agent, MIB Compiler and Code generator, configuration API properties, and protocol stack independent functions.

### 3.1 How To Build the Agent

The SNMP agent can be built as SNMPv1-only agent, or SNMPv1/v2c bilingual agent, or full SNMP v1/v2c/v3 trilingual agent.

If the agent is built as SNMPv1, it will recognize and respond only to SNMPv1 messages, the error-status field being used as defined in RFC 1157, and issue only SNMPv1 traps.

If the agent is built as SNMPv1/v2c, it will recognize SNMPv1 and SNMPv2c messages and respond accordingly. The error-status field in SNMPv1 GetResponse messages conforms to the definitions in RFC 1157, and in SNMPv2c responses to those in RFC 1905. The traps are issued both in SNMP1 and SNMPv2c formats.

If the agent is built as SNMPv1/v2c/v3, it will recognize SNMPv1, SNMPv2c and SNMPv3 messages and respond accordingly. The error-status field in SNMPv1 responses conforms to the definitions in RFC 1157, and in SNMPv2c and SNMPv3 responses to those in RFC 1905. The traps are issued in SNMP1, SNMPv2c and SNMPv3 formats.

To enable SNMP agent functionality with the Treck TCP/IP stack, you must edit your trsystem.h file, and uncomment exactly one of the following macro definitions corresponding to whether you want SNMPv1-only agent, or SNMPv1/v2c bilingual agent, or full SNMP v1/v2c/v3 trilingual agent:

```

/*
 * Uncomment the following line for SNMPv1 only.
 * (SNMP option only.)
 */
/* #define TM_SNMP_VERSION 1 */

/*
 * Uncomment the following line for SNMPv1 and SNMPv2c (bilingual).
 * (SNMP option only.)
 */
/* #define TM_SNMP_VERSION 2 */

/*
 * Uncomment the following line for SNMPv1, SNMPv2c and SNMPv3 (trilingual).
 * (SNMP option only.)
 */
/* #define TM_SNMP_VERSION 3 */

```

Additionally, if you want the SNMP Agent to support GETNEXT requests with the MIB-II tables ipNetToMediaTable (ARP cache), ipRouteTable (Routing table), udpTable (UDP listening sockets) and tcpConnTable (TCP sockets), then you will also need to #define the TM\_SNMP\_CACHE macro in your trsystem.h file. After you make these changes to your trsystem.h file, you will need to do a full recompile of both the Treck TCP/IP stack and the SNMP Agent source code.

### 3.1.1 Building the SNMP Agent Library (MS-DOS only)

If you are building the Treck TCP/IP stack on a MS-DOS platform, then you may be using the provided MS-DOS batch files described in the Treck TCP/IP user documentation (refer to chapter 4 “Integrating Treck Real-Time Protocols Into Your Environment”, section “Step 3 Creating the Build Command (.BAT) Files for a DOS Environment and Building the Library”). A couple of new batch files are provided with the SNMP Agent to assist you in compiling the SNMP Agent code and building a SNMP Agent object library to link your application against. These batch files are located in the treckbin subdirectory of the SNMP Agent product CD.

#### **snmpd.bat**

You execute snmpd.bat in the directory that has the SNMP Agent source files, and it compiles the SNMP Agent source code. This batch file takes two parameters:

- Parameter 1: the name of the tier 1 .BAT file to use. This is the “compiler utility primitive” described by the Treck TCP/IP user documentation.
- Parameter 2: the optional #define to pass to the compile on the command line.

#### **snmpdlib.bat**

You execute snmpdlib.bat in the directory that has the SNMP Agent source files (also, the compiled object files created by snmpd.bat), and it inserts the compiled object files into the SNMP Agent object library. This batch file takes two parameters:

- Parameter 1: the name of the tier 1 .BAT file to use. This is the “library utility primitive” described by the Treck TCP/IP user documentation.
- Parameter 2: the name of the library to create.

## 3.2 Running the Agent

The user can start or stop the agent by calling tfSnmpdMain or tfSnmpdStop respectively. The user needs to #include the header file trapi.h in their code which calls these APIs.

### 3.2.1 tfSnmpdMain

The user calls tfSnmpdMain to initialize the agent and start listening for the incoming messages. tfSnmpdMain can be either blocking or non-blocking, as specified by the last parameter.

#### 3.2.1.1 Blocking Mode

tfSnmpdMain should be called from a task. tfSnmpdMain will not return unless a system error occurs or is stopped by tfSnmpdStop, and will wait for and respond to incoming messages, and execute the code in the context of the calling task. Choose blocking mode if you are using a RTOS/Kernel and prefer to run the SNMP Agent as a separate task.

#### 3.2.1.2 Non-Blocking Mode

tfSnmpdMain will return immediately after initialization, and reception, processing and response to SNMP messages occurs afterwards in a socket callback function. Choose non-blocking mode if you do not have a RTOS/Kernel.

### 3.2.2 tfSnmpdStop

The user calls tfSnmpdStop to close the SNMP sockets and kill the agent. tfSnmpdStop does the reverse of initializing the agent, i.e. any memory that was allocated for the SNMP Agent during initialization is now freed.

### 3.3 Macros

A number of SNMP configuration parameters must be defined prior to starting the agent. These parameters include trap configuration parameters, SNMP port and communities information for SNMP v1/v2c configuration. If the SNMPv3 agent is built as explained above, then additional parameters related to five tables must be also defined. The default values are provided for ALL parameters. However, if the user decides to change some or all configuration parameters, he/she can do so via Configuration API functions defined in the next Chapter.

The default values for the configuration parameters are contained in `trcfg.h` and `tregstr.h`

SNMP protocol recommends UDP port 161 as a port on which the agent listens to the requests. The following line in `trcfg.h` defines the SNMP default port:

```
#define TM_SNMP_DEF_PORT      161
```

However, for the security or other reasons, the user might want to set the SNMP port to a different value. To do so, he/she must use the function `tfSetSnmpPort()` defined in the next Chapter.

---

**NOTE:**

1. *If you set this port to the value other than default, the manager(s) communicating with the agent must be aware of the new port value. Most managers use port 161 as a default.*
  2. *This function may be called ONLY prior to starting the agent, and NEVER called while the agent is running.*
- 

SNMP protocol does not specify a number of communities an SNMP agent might use for the message authentication, nor does it specify that different communities are to be used for retrieval and modification requests. This agent defines a maximum number of 'read' and 'write' communities and a maximum length of the human readable community string in `tregstr.h`:

```
#define TM_SNMP_MAX_NUM_READ_COMMUNITIES  5
#define TM_SNMP_MAX_NUM_WRITE_COMMUNITIES 5
#define TM_SNMP_MAX_COMMUNITY_LEN        24
```

Note that no Configuration API function is provided for changing these maximum numbers. The defined values are large enough to accommodate any needs. However, if the need arises to increase these values, the the user should simply redefine them in `tregstr.h` and rebuild the agent.

By default, the agent starts with one 'read' community and one 'write' community. They are defined in `trcfg.h` as:

```
#define TM_SNMP_READ_COMMUNITY    "public"
#define TM_SNMP_WRITE_COMMUNITY  "private"
```

The user can add, delete, or modify the community names by using `tfAddCommunity()`, `tfDeleteCommunity()` and `tfModifyCommunity()` defined in the next Chapter. The only limitation is, if only one community is left in the list, the user is not allowed to delete it.

SNMP protocol does not specify a number of trap sinks (the network entities to which the agent sends the traps) or their IP addresses. This agent defines a maximum number of trap sinks in `tregstr.h` as follows:

```
#define TM_SNMP_MAX_NUM_TRAP_SINKS  5
```

Note, that, as in case of communities, this number is not changeable via Configuration API, only through manual modification and rebuilding the agent.

The trap message contains the fields which are defined in `trcfg.h`:

```
#define TM_SNMP_TRAP_DEST_ADDR    "192.168.1.2"
#define TM_SNMP_TRAP_DEF_PORT     162
#define TM_SNMP_TRAP_COMMUNITY   "trap comm"
```

and `trregstr.h`:

```
#define TM_DEFAULT_ENTERPRISE_NUMBER    29999
#define TM_DEFAULT_ENTERPRISE_LENGTH   8
```

The user can add, delete, or modify the trap information by using `tfAddTrapEntry()`, `tfDeleteTrapEntry()` and `tfModifyTrapEntry()` defined in the next Chapter. The only limitation is, if only one trap entry is left in the list, the user is not allowed to delete it.

The enterprise entries in `trregstr.h` are only relevant for SNMPv1 traps.

By default, the agent starts with one trap entry.

The SNMP MIB-II system group, implementation of which is mandatory for all agents, contains seven objects. The default values for all objects except `sysUpTime` are defined in `trcfg.h`:

```
const char tvSnmpSysDescription[] = "Treck SNMP Agent";
const oid tvSnmpSysObjid[] = { 1, 3, 6, 1, 4, 1, 29999, 1 };
const int tvSnmpSysObjidLen =
    sizeof (tvSnmpSysObjid) / sizeof (oid);
const char tvSnmpSysContact[] = "Treck HQ";
const char tvSnmpSysName[] = "Treck, Inc.";
const char tvSnmpSysLocation[] = "San Francisco, CA USofA";
const unsigned short tvSnmpSysServices = 72;
```

Three objects, `sysContact`, `sysName` and `sysLocation` contain access clauses of read-write, i.e. their instances can be changed via SNMP requests, hence the Configuration API functions for their modifications are not provided. The other three members, `sysDescription`, `sysObjectID` and `sysServices`, contain access clauses of read-only, and can not be modified through SNMP requests. The modification functions, `tfModifySysDescr()`, `tfModifySysObjId()` and `tfModifySysServices()` can be used to modify these objects instances, and are described in the next Chapter.

As mentioned above, five tables must be initialized to build SNMPv3 agent. These tables are:

- `usmUserTable`
- `vacmContextTable`
- `vacmSecurityToGroupTable`
- `vacmAccessTable`
- `vacmViewTreeFamilyTable`

These tables are defined in RFC 2574 and RFC 2575 and contain objects identifying parameters of User-Based Security Model (`usmUserTable`) and View-Based Access Control Model (the rest). Though most of the objects in these tables contain access clauses of read-create, and therefore their instances can be modified via SNMP requests, the Configuration API functions for creation (modification) and deletion of entries in these tables are provided for the user convenience. These functions can be called either before starting the agent or while the agent is running.

Upon starting, the agent checks the values of pointers `snmpUsmUserEntryStart`, `snmpVacmContextEntryStart`, `snmpVacmSec2GroupEntryStart`, `snmpVacmAccessEntryStart`, and `snmpVacmViewTreeFamilyEntryStart` contained in `snmpInitTables` within `ttSnmpd`, defined in `trregstr.h`. If any of the pointers is non-zero, the agent concludes the corresponding table is already initialized and skips default initialization. Otherwise, the agent initializes the corresponding table entries to their default values, defined in `trv3init.c`:

Two default entries are defined for usmUserTable objects –

usmUserName	“initial”
usmUserSecurityName	“INITIAL”
usmUserCloneFrom	1.3.6.1.4.1.115.0
usmUserAuthProtocol	usmHMACMD5AuthProtocol
usmUserAuthKeyChange	“Treck”
usmUserOwnAuthKeyChange	“Treck USA”
usmUserPrivProtocol	usmDESPrivProtocol
usmUserPrivKeyChange	“Treck”
usmUserOwnPrivKeyChange	“Treck USA”
usmUserPublic	“Public”
usmUserStorageType	readOnly
usmUserStatus	active

and

usmUserName	“initial1”
usmUserSecurityName	“securityName”
usmUserCloneFrom	1.3.6.1.4.1.115.0
usmUserAuthProtocol	usmNoAuthProtocol
usmUserAuthKeyChange	“auth key”
usmUserOwnAuthKeyChange	“own auth key”
usmUserPrivProtocol	usmNoPrivProtocol
usmUserPrivKeyChange	“privacy key”
usmUserOwnPrivKeyChange	“ownpriv key”
usmUserPublic	“private”
usmUserStorageType	readOnly
usmUserStatus	active

For the definitions of usmHMACMD5AuthProtocol, usmDESPrivProtocol, usmNoAuthProtocol and usmNoPrivProtocol see RFC 2574, for definitions of readOnly and active see RFC 1903.

Three default entries are defined for vacmContextTable object –

vacmContextName	Empty octet string
vacmContextName	“context_1”
vacmContextName	“context_2”

Two default entries are defined for vacmSecurityToGroupTable objects –

vacmSecurityModel	User-Based Security Model (USM)	vacmSecurityName	“initial”
vacmGroupName	“group 1”		
vacmSecurityToGroupStorageType	readOnly		
vacmSecurityToGroupStatus	active		

and

vacmSecurityModel	User-Based Security Model (USM)	vacmSecurityName	“Treck”
vacmGroupName	“group 2”		
vacmSecurityToGroupStorageType	readOnly		
vacmSecurityToGroupStatus	active		

For the definition of User-Based Security Model (USM) see RFC 2571.

Two default entries are defined for vacmAccessTable objects –

vacmAccessContextPrefix	“context_2”	vacmAccessSecurityModel	User-Based Security
Model (USM)		vacmAccessSecurityLevel	noAuthNoPriv
vacmAccessContextMatch	exact	vacmAccessReadViewName	“read view”
vacmAccessWriteViewName	“write view”	vacmAccessNotifyViewName	“notify view”
vacmAccessStorageType	readOnly	vacmAccessStatus	active

and

vacmAccessContextPrefix	Empty octet string	vacmAccessSecurityModel	User-Based
Security Model (USM)		vacmAccessSecurityLevel	noAuthNoPriv
vacmAccessContextMatch	prefix	vacmAccessReadViewName	“read view”
vacmAccessWriteViewName	“write view”	vacmAccessNotifyViewName	“notify view”
vacmAccessStorageType	readOnly	vacmAccessStatus	active

In addition, vacmGroupName which is a part of the vacmAccessTable index, is defaulted to “group 1” and “group 2” correspondingly.

For the definition of noAuthNoPriv see RFC 2571, for the definitions of exact and prefix see RFC 2575.

One default entry is defined for vacmViewTreefamilyTable objects –

<code>vacmViewtreeFamilyViewName</code>	<code>"read view"</code>	<code>vacmViewtreeFamilySubtree</code>	1.3.6.1.2.1.3.1.5
<code>vacmViewtreeFamilyMask</code>	<code>0xf000</code>		
<code>vacmViewtreeFamilyType</code>	<code>included</code>	<code>vacmViewtreeFamilyStorageType</code>	<code>readOnly</code>
<code>vacmViewtreeFamilyStatus</code>	<code>active</code>		

For the definition of `included` see RFC 2575.

Two secret passwords used in authentication and privacy operation are defined in `trcfg.h`:

```
#define TM_SNMP_USER_AUTH_PSWD      "TreckUSA"
#define TM_SNMP_USER_PRIVACY_PSWD   "TreckInc"
```

These are default values. Upon starting, the agent checks the values of pointers `snmpUserAuthPswd` and `snmpUserPrivacyPswd` contained in `snmpInitTables` within `ttSnmpd`, defined in `trregstr.h`. If any of the pointers is non-zero, the agent concludes the corresponding string is already initialized and skips default initialization. Otherwise, the agent initializes the corresponding strings to their default values. The user may modify these default strings by using `tfModifyAuthPassword()` and `tfModifyPrivacyPassword()` prior to starting the agent. Note, that maximum lengths of `snmpUserAuthPswd` and `snmpUserPrivacyPswd` are defined in `trregstr.h` as:

```
#define TM_SNMP_MAX_USER_AUTH_PSWD_LEN      24
#define TM_SNMP_MAX_USER_PRIVACY_PSWD_LEN   24
```

Note that no Configuration API function is provided for changing these maximum lengths. However, if the need arises to increase these values, the the user should simply redefine them in `trregstr.h` and rebuild the agent.

## 3.4. SNMPv3 initialization

### 3.4.1. Overview

The SNMPv3 framework consists of 11 tables defined in the USM (User-Based Security Model) MIB (RFC 2574), the VACM (View-Based Access Control Model) MIB (RFC 2575), the community MIB (RFC 2576), the target MIB (RFC 2573) and the notification MIB (RFC 2573).

- `usmUserTable`
- `vacmContextTable`
- `vacmSecurityToGroupTable`
- `vacmAccessTable`
- `vacmViewTreeFamilyTable`
- `snmpCommunityTable`
- `snmpTargetAddrTable`
- `snmpTargetParamsTable`
- `snmpNotifyTable`
- `snmpNotifyFilterTable`
- `snmpNotifyFilterProfileTable`

Though most of the objects in these tables contain access clauses of read-create, and therefore their instances can be modified via SNMP SET requests, the Configuration API functions for creation (modification) and deletion of entries in these tables are provided for the user convenience. These functions can be called either before starting the agent or while the agent is running.

Upon starting, the agent checks the values of pointers `snmpUsmUserEntryStart`, `snmpVacmContextEntryStart`, `snmpVacmSec2GroupEntryStart`, `snmpVacmAccessEntryStart`, `snmpVacmViewTreeFamilyEntryStart`, `snmpCommunityTableEntryStart`, `snmpTargetAddrTableEntryStart`, `snmpTargetParamsTableEntryStart`, `snmpNotifyTableEntryStart`, `snmpNotifyFilterProfileTableEntryStart`, `snmpNotifyFilterTableEntryStart` contained in `snmpInitTables` within `ttSnmpd`, defined in `trregstr.h`. If any of the pointers is non-zero, the agent concludes the corresponding table is already initialized and skips default initialization. Otherwise, the agent initializes the corresponding table entries to their default values, defined in `trv3init.c`.

The entries defined in those tables must be ordered in lexicographic order the way they would be ordered when walking the MIB.



```
TM_SNMP_STORAGE_TYPE_PERMANENT, TM_SNMP_ROWSTATUS_ACTIVE
},
/* This row must stay at the end */
{"", 0, "", 0, {0}, 0, {0}, 0, "", 0, "", 0, {0}, 0,
 "", 0, "", 0, "", 0, -1, 0 }
```

User "initial" is a user configured with "authPriv" authentication level using MD5 for authentication and DES for privacy. Its authentication password is "authkey", its privacy password is "privkey".

User "initial1" is a user configured with "noAuthNoPriv" authentication level.

### 3.4.3. vacmContextTable

This table (RFC2574) defines the SNMP contexts supported by the agent. A SNMPv3 request is always handled within a particular SNMP context (included in the packet header). A different set of users and groups can be defined for different SNMP contexts. Most applications will run the agent with only one context, the default "" (empty string).

*Data Structure:*

```
typedef struct tsSnmPVacmContextInit
{
    char        svciName[32];
    int         svciNameLen;
} ttSnmPVacmContextInit;
```

*Initialization table in trv3init.c:* tvSnmPVacmContextInit[]

Must be sorted by svciNameLen and svciName.

*Configuration example:*

```
const ttSnmPVacmContextInit tvSnmPVacmContextInit[] =
{
    {"", 0},
    {"context_1", 9},
    {"", -1}
};
```

Two contexts have been defined: "" (the default context) and "context\_1".

### 3.4.4. vacmSecurityToGroupTable

This table defined in RFC2575 is used to map users of the usmUserTable with groups of the vacmAccessTable. Once the packet is correctly authenticated and eventually decrypted using the information contained in the usmUserTable, the user read/right permissions contained in the vacmAccessTable are located by consulting the vacmSecurityToGroupTable.

*Data structure:*

```
typedef struct tsSnmPVacmSec2GroupInit
{
    int          svsgModel;
    char         svsgName[32];
    int          svsgNameLen;
    char         svsgGroupName[32];
    int          svsgGroupNameLen;
    int          svsgStorageType;
    int          svsgStatus;
} ttSnmPVacmSec2GroupInit;
```

*Initialization table in trv3init.c:* tvSnmPVacmSec2GroupInit[]

Must be sorted by svsgModel, svsgNameLen and svsgName

*Configuration example:*

```
const ttSnmPVacmSec2GroupInit tvSnmPVacmSec2GroupInit[] =
{
    {TM_SNMP_SECMODEL_V1, "publicuser", 10, "group 1", 7,
    TM_SNMP_STORAGETYPE_PERMANENT, TM_SNMP_ROWSTATUS_ACTIVE},
    {TM_SNMP_SECMODEL_V2C, "publicuser", 10, "group 1", 7,
    TM_SNMP_STORAGETYPE_PERMANENT, TM_SNMP_ROWSTATUS_ACTIVE},
    {TM_SNMP_SECMODEL_USM, "publicuser", 10, "group 1", 7,
    TM_SNMP_STORAGETYPE_PERMANENT, TM_SNMP_ROWSTATUS_ACTIVE},
/* This row must stay at the end */
    {-1, "", 0, "", 0, -1, 0}
};
```

In this example, the user "publicuser" is bound to the group "group 1" for all security models (V1, V2c and USM).

### 3.4.5. vacmAccessTable

This table contains VACM groups as defined in RFC 2575. A group consists of a set of access rights defined with mib-views.

*Data structure:*

```
typedef struct tsSnmVAcMAccessInit
{
    char        svaiGroupName[32];
    int         svaiGroupNameLen;
    char        svaiPrefix[32];
    int         svaiPrefixLen;
    int         svaiModel;
    int         svaiLevel;
    int         svaiMatch;
    char        svaiReadName[32];
    int         svaiReadNameLen;
    char        svaiWriteName[32];
    int         svaiWriteNameLen;
    char        svaiNotifyName[32];
    int         svaiNotifyNameLen;
    int         svaiStorageType;
    int         svaiStatus;
} ttSnmVAcMAccessInit;
```

*Initialization table in trv3init.c:* tvSnmVAcMAccessInit[]

Must be sorted by svaiPrefixLen, svaiPrefix, svaiModel and svaiLevel.

*Configuration example:*

```
const ttSnmVAcMAccessInit tvSnmVAcMAccessInit[] =
{
    {"publicgroup", 11, "", 0,
     TM_SNMP_SECMODEL_V1, TM_SNMP_SECLEVEL_NOAUTHNOPRIV, 1,
     "allmib", 6, "none", 4, "notify view", 11,
     TM_SNMP_STORAGETYPE_PERMANENT, TM_SNMP_ROWSTATUS_ACTIVE},
    {"publicgroup", 11, "", 0,
     TM_SNMP_SECMODEL_V2C, TM_SNMP_SECLEVEL_NOAUTHNOPRIV, 1,
     "allmib", 6, "none", 4, "notify view", 11,
     TM_SNMP_STORAGETYPE_PERMANENT, TM_SNMP_ROWSTATUS_ACTIVE},
    {"publicgroup", 11, "", 0,
     TM_SNMP_SECMODEL_USM, TM_SNMP_SECLEVEL_NOAUTHNOPRIV, 1,
     "allmib", 6, "none", 4, "notify view", 11,
     TM_SNMP_STORAGETYPE_PERMANENT, TM_SNMP_ROWSTATUS_ACTIVE},

    {"privategroup", 12, "", 0,
     TM_SNMP_SECMODEL_USM, TM_SNMP_SECLEVEL_AUTHNOPRIV, 1,
     "allmib", 6, "allmib", 6, "notify view", 11,
     TM_SNMP_STORAGETYPE_PERMANENT, TM_SNMP_ROWSTATUS_ACTIVE},
    /* This row must stay at the end */
    {"", 0, "", 0, -1, 0, 0, "", 0, "", 0, "", 0, 0, 0}
};
```

In this example, users who belong to group “publicgroup” can read (GET/GETNEXT) the objects of the mib defined in the mib-view “allmib” and can write (SET) the objects of the mib defined by the mib-view “none”. Users belonging to the group “privategroup” can both get and set objects of the view “allmib”. Users of this group can however only perform request if they are authenticated. Both groups are defined within the default SNMP context (“”).



### 3.4.7. snmpCommunityTable

This table (RFC2576) defines a way to use the View-based access control model (preliminary defined for SNMPv3) for SNMPv1 and SNMPv2c requests. It maps a v1 or v2 community to a SNMPv3 user. The same access rights that apply to the USM user apply to the community. The equivalent user need not exist in the usmUserTable but must be referenced in the vacmSecurityToGroupTable.

*Data structure:*

```
typedef struct tsSnmpCommunityTableInit
{
    char    commIndex[32];
    tt32Bit commIndexLen;
    char    commName[32];
    tt32Bit commNameLen;
    char    commSecName[32];
    tt32Bit commSecNameLen;
    int     commStorageType;
    int     commStatus;
} ttSnmpCommunityTableInit;
```

*Initialization table in trv3init.c:* tvSnmpCommunityTableInit[]

Must be sorted by commIndexLen and commIndex.

*Configuration example:*

```
const ttSnmpCommunityTableInit tvSnmpCommunityTableInit[] =
{
    {"publiccom", 9, "public", 6, "publicuser", 10,
     TM_SNMP_STORAGETYPE_NONVOLATILE, TM_SNMP_ROWSTATUS_ACTIVE},
    {"privatecom", 10, "private", 7, "privateuser", 11,
     TM_SNMP_STORAGETYPE_NONVOLATILE, TM_SNMP_ROWSTATUS_ACTIVE},
    /* This row must stay at the end */
    {"", 0, "", 0, "", 0, -1, 0}
};
```

In this example, the entry indexed by "publiccom" defines a mapping between the community name "public" and the v3 User "publicuser". The "privatecom" entry maps the "private" community name to the user "privateuser". When a V1 (V2) packet is received with the community name set to "private" for instance, the agent will search in the vacmSecurityToGroupTable for an entry corresponding to "privateuser" with security model set to V1 (V2) and will apply the access rights associated with the user and the group it belongs to (looking at the vacmAccessTable).

### 3.4.8. snmpNotifyTable

This table (RFC2573) defines an SNMP notification (trap).

*Data structure:*

```
typedef struct ttSnmpNotifyTableInit
{
    char    snmpNotifyName[32];
    tt32Bit snmpNotifyNameLen;
    char    snmpNotifyTag[32];
    tt32Bit snmpNotifyTagLen;
    tt32Bit snmpNotifyType;
    int     snmpNotifyStorageType;
    int     snmpNotifyStatus;
} ttSnmpNotifyTableInit;
```

*Initialization table in trv3init.c:* tvSnmpNotifyTableInit[]

Sorted only by snmpNotifyName.

*Configuration example:*

```
const ttSnmpNotifyTableInit tvSnmpNotifyTableInit[] =
{
    {"defaultnotification", 19, "tag1", 4,
     TM_SNMP_NOTIFYTYPE_TRAP,
     TM_SNMP_STORAGE_TYPE_NONVOLATILE, TM_SNMP_ROWSTATUS_ACTIVE},
    /* This row must stay at the end */
    {"", 0, "", 0, 0, -1, 0}
};
```

This example defines a trap identified by “defaultnotification”. The destination host and trap parameters will be defined in the snmpTargetAddrTable, at the entry “tag1”.

### 3.4.9. snmpNotifyFilterTable

The notifyFilterTable (RFC 2573) contains a list of sub-trees used to filter outgoing notifications the same way mib-views are used to filter incoming SNMP requests.

Data structure:

```
typedef struct tsSnmpNotifyFilterTableInit
{
    char    snmpNotifyFilterProfileName[32];
    tt32Bit snmpNotifyFilterProfileNameLen;
    oid     snmpNotifyFilterSubtree[32];
    tt32Bit snmpNotifyFilterSubtreeLen;
    char    snmpNotifyFilterMask[32];
    tt32Bit snmpNotifyFilterMaskLen;
    tt32Bit snmpNotifyFilterType;
    int     snmpNotifyFilterStorageType;
    int     snmpNotifyFilterRowStatus;
} ttSnmpNotifyFilterTableInit;
```

*Initialization table in trv3init.c:* tvSnmpNotifyFilterProfileTableInit

Must be sorted by snmpNotifyNameLen and snmpNotifyName.

*Configuration example:*

```
const ttSnmpNotifyFilterTableInit tvSnmpNotifyFilterTableInit[] =
{
    {"public profile", 14,
     {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
     1 * sizeof(oid), {0x00, 0x00}, 2, TM_SNMP_VACM_VIEW_INCLUDED,
     TM_SNMP_STORAGE_TYPE_NONVOLATILE, TM_SNMP_ROWSTATUS_ACTIVE},
    /* This row must stay at the end */
    {"", 0, {0}, 0, "", 0, 0, -1, 0}
};
```

### 3.4.10. snmpTargetAddrTable

This table (RFC2473) contains a list of transport addresses to be used in the generation of the SNMP notification.

*Data structure:*

```
typedef struct tsSnmpTargetAddrTableInit
{
    char    snmpTargetAddrName[32];
    tt32Bit snmpTargetAddrNameLen;
    tt32Bit snmpTargetAddrTDomain;
    char    snmpTargetAddr[64];
    tt32Bit snmpTargetAddrPort;
    tt32Bit snmpTargetAddrTimeout;
    tt32Bit snmpTargetRetryCount;
    char    snmpTargetTagList[32];
    tt32Bit snmpTargetTagListLen;
    char    snmpTargetAddrParams[32];
    tt32Bit snmpTargetAddrParamsLen;
    int     snmpTargetAddrStorageType;
    int     snmpTargetAddrRowStatus;
} ttSnmpTargetAddrTableInit;
```

*Initialization table in trv3init.c:* tvSnmpTargetAddrTableInit[]

Must be sorted by targetAddrName, targetAddrTDomain

*Configuration example:*

```
const ttSnmpTargetAddrTableInit tvSnmpTargetAddrTableInit[] =
{
    {"target1", 7, AF_INET, "192.168.1.2", 162, 0, 0,
     "tag1", 19, "paramv1", 7,
     TM_SNMP_STORAGETYPE_NONVOLATILE, TM_SNMP_ROWSTATUS_ACTIVE},
    /* This row must stay at the end */
    {"", 0, 0, "", 0, 0, 0, "", 0, "", 0, -1, 0}
};
```

The example above defines a target at the IPv4 address 192.168.1.2 on port 162. Entries in the snmpNotifyTable will reference entries of this table using the tag parameter "tag1". The value "paramv1" identifies a row of the snmpTargetParamsTable used when generating a message to this target.

### 3.4.11. snmpNotifyFilterProfileTable

This table (RFC 2473) associates a notification filter profile with a particular set of target parameters.

*Data Structure:*

```
typedef struct tsSnmpNotifyFilterProfileTableInit
{
    char    snmpNotifyTargetParamsName[32];
    tt32Bit snmpNotifyTargetParamsNameLen;
    char    snmpNotifyFilterProfileName[32];
    tt32Bit snmpNotifyFilterProfileNameLen;
    int     snmpNotifyFilterProfileStorType;
    int     snmpNotifyFilterProfileRowStatus;
} ttSnmpNotifyFilterProfileTableInit;
```

*Initialization table in trv3init.c:* tvSnmpNotifyFilterProfileTableInit[]

*Configuration example:*

```
const ttSnmpNotifyFilterProfileTableInit tvSnmpNotifyFilterProfileTableInit[] =
{
    {"paramv1", 7, "public profile", 14,
     TM_SNMP_STORAGETYPE_NONVOLATILE, TM_SNMP_ROWSTATUS_ACTIVE},
    /* This row must stay at the end */
    {"", 0, "", 0, -1, 0}
};
```

This example provides a mapping between the targetParamsName "paramv1" and the profile "public profile" defined in the snmpNotifyFilterTable.

### 3.4.12. snmpTargetParamsTable

The notification targets are defined in the snmpTargetAddrTable (RFC 2473). The snmpTargetParamsTable provides additional information required to generate a notification and send it to a specific target.

*Data Structure:*

```
typedef struct tsSnmpTargetParamsTableInit
{
    char    snmpTargetParamsName[32];
    tt32Bit snmpTargetParamsNameLen;
    tt32Bit snmpTargetParamsMPModel;
    tt32Bit snmpTargetParamsSecurityModel;
    char    snmpTargetParamsSecurityName[32];
    tt32Bit snmpTargetParamsSecurityNameLen;
    tt32Bit snmpTargetParamsSecurityLevel;
    int     snmpTargetParamsStorageType;
    int     snmpTargetParamsRowStatus;
} ttSnmpTargetParamsTableInit;
```

*Initialization table in trv3init.c:* tvSnmpTargetParamsTableInit[]

Must be sorted by snmpTargetParamsName

*Configuration example:*

```
const ttSnmpTargetParamsTableInit tvSnmpTargetParamsTableInit[] =
{
    {"paramv1", 7,
     TM_SNMP_MPMODEL_V1, TM_SNMP_SECLEVEL_NOAUTHNOPRIV,
     "public", 6, 2,
     TM_SNMP_STORAGE_TYPE_NONVOLATILE, TM_SNMP_ROWSTATUS_ACTIVE},
    /* This row must stay at the end */
    {"", 0, 0, 0, "", 0, 0, -1, 0}
};
```

This example indicates the notifications defined with the parameter “paramv1” will be sent in a V1 trap PDU, using the community name “public”.

### 3.4.13. Default Configuration

Treck SNMP is initialized by default with the following configuration:

#### *usmUserTable* default Values:

<i>usmUserName</i>	“initial”	“initial1”
<i>usmUserSecurityName</i>	“INITIAL”	“securityName”
<i>usmUserCloneFrom</i>	1.3.6.1.4.1.115.0	1.3.6.1.4.1.115.0
<i>usmUserAuthProtocol</i>	usmHMACMD5AuthProtocol	usmNoAuthProtocol
<i>usmUserAuthKeyChange</i>	“Treck”	“auth key”
<i>usmUserOwnAuthKeyChange</i>	“Treck Inc”	“own auth key”
<i>usmUserPrivProtocol</i>	usmDESPrivProtocol	usmNoPrivProtocol
<i>usmUserPrivKeyChange</i>	“Treck”	“privacy key”
<i>usmUserOwnPrivKeyChange</i>	“Treck USA”	“ownpriv key”
<i>usmUserPublic</i>	“Public”	“private”
<i>usmUserStorageType</i>	readOnly	readOnly
<i>usmUserStatus</i>	Active	Active

#### *vacmContextTable* default Values:

<i>vacmContextName</i>	“”	“context_1”	“context_2”
------------------------	----	-------------	-------------

#### *vacmSecurityToGroupTable* default Values:

<i>vacmSecurityModel</i>	publicuser	privateuser	publicuser
<i>vacmSecurityName</i>	V1	V1	V2c
<i>vacmGroupName</i>	group 1	group 2	group 1
<i>vacmSecurityToGroupStorageType</i>	permanent	permanent	permanent
<i>vacmSecurityToGroupStatus</i>	Active	Active	Active
<i>vacmSecurityModel</i>	privateuser	initial	publicuser
<i>vacmSecurityName</i>	V2c	USM	USM
<i>vacmGroupName</i>	group 2	group 1	group 1
<i>vacmSecurityToGroupStorageType</i>	permanent	permanent	permanent
<i>vacmSecurityToGroupStatus</i>	Active	Active	Active

#### *vacmAccessTable* default values:

<i>vacmAccessGroupName</i>	“group 1”	“group 1”	“group 1”
<i>vacmAccessContextPrefix</i>	“”	“”	“”
<i>vacmAccessSecurityModel</i>	V1	V2	USM
<i>vacmAccessSecurityLevel</i>	noAuthNoPriv	noAuthNoPriv	NoAuthNoPriv
<i>vacmAccessContextMatch</i>	exact	exact	exact
<i>vacmAccessReadViewName</i>	“allmib”	“allmib”	“allmib”
<i>vacmAccessWriteViewName</i>	“none”	“none”	“none”
<i>vacmAccessNotifyViewName</i>	“notify view”	“notify view”	“notify view”
<i>vacmAccessStorageType</i>	permanent	permanent	permanent
<i>vacmAccessStatus</i>	Active	Active	Active
<i>vacmAccessGroupName</i>	“group 2”	“group 2”	“group 2”
<i>vacmAccessContextPrefix</i>	“”	“”	“”
<i>vacmAccessSecurityModel</i>	V1	V2	USM
<i>vacmAccessSecurityLevel</i>	noAuthNoPriv	noAuthNoPriv	NoAuthNoPriv
<i>vacmAccessContextMatch</i>	exact	exact	Exact
<i>vacmAccessReadViewName</i>	“allmib”	“allmib”	“allmib”
<i>vacmAccessWriteViewName</i>	“allmib”	“allmib”	“allmib”
<i>vacmAccessNotifyViewName</i>	“notify view”	“notify view”	“notify view”
<i>vacmAccessStorageType</i>	permanent	permanent	permanent
<i>vacmAccessStatus</i>	Active	Active	Active

**vacmViewTreeFamilyTable default value:**

vacmViewTreeFamilyViewName	“mib2”	“none”	“all mib”
vacmViewTreeFamilySubtree	1.3.6.1.2.1.	1.	1.
vacmViewTreeFamilyMask	0xFC	0x00	0x00
vacmViewTreeFamilyType	included	excluded	included
vacmViewTreeFamilyStorageType	permanent	permanent	permanent
vacmViewTreeFamilyStatus	Active	Active	Active

**snmpCommunityTable default value:**

snmpCommunityIndex	“publiccom”	“privatecom”
snmpCommunityName	“public”	“private”
snmpCommunitySecurityName	“publicuser”	“privateuser”
snmpCommunityContextEngineID	N/A*	N/A*
snmpCommunityContextName	“”	“”
snmpCommunityTransportTag	N/A*	N/A*
snmpCommunityStorageType	permanent	permanent
snmpCommunityStatus	Active	Active

**snmpNotifyTable default values:**

snmpNotifyName	“defaultnotification”
snmpNotifyTag	“defaultnotification”
snmpNotifyType	Trap
snmpNotifyStorageType	nonvolatile
snmpNotifyRowStatus	Active

**snmpNotifyFilterProfileTable default values:**

snmpNotifyFilterProfileName	“paramv1”	“paramv2”	“paramv3”
snmpNotifyFilterProfileStorType	nonVolatile	nonVolatile	nonVolatile
snmpNotifyFilterProfileRowStatus	Active	Active	Active

**snmpNotifyFilterTable default values:**

snmpNotifyFilterProfileName	“public profile”
snmpNotifyFilterSubtree	1.
snmpNotifyFilterMask	0x00
snmpNotifyFilterType	included
snmpNotifyFilterStorageType	nonVolatile
snmpNotifyFilterRowStatus	Active

**snmpTargetAddrTable default values:**

snmpTargetAddrName	“target1”	“target2”	“target3”
snmpTargetAddrTDomain	UdplpV4	UdplpV4	UdplpV4
snmpTargetAddrTAddress	“192.168.1.2:162”	“192.168.1.2:162”	“192.168.1.2:162”
snmpTargetAddrTimeout	0	0	0
snmpTargetAddrRetryCount	0	0	0
snmpTargetAddrTagList	“defaultnotification”	“defaultnotification”	“defaultnotification”
snmpTargetAddrParams	“paramv1”	“paramv2”	“paramv3”
snmpTargetAddrStorageType	nonVolatile	nonVolatile	nonVolatile
snmpTargetAddrRowStatus	Active	Active	Active

**snmpTargetParamsTable default values:**

snmpTargetParamsName	“paramv1”	“paramv2”	“paramv3”
snmpTargetParamsMPModel	V1	V2	V3
snmpTargetParamsSecurityModel	V1	V2	USM
snmpTargetParamsSecurityName	“public”	“public”	“public”
snmpTargetParamsSecurityLevel	noAuthNoPriv	noAuthNoPriv	noAuthNoPriv
snmpTargetParamsStorageType	nonVolatile	nonVolatile	nonVolatile
snmpTargetParamsRowStatus	Active	Active	Active

## 3.5 Running treck SNMP without Treck TCP/IP

The Treck SNMP agent has been designed primarily to run on top of the Treck TCP/IP stack and is by default integrated to the Treck TCP/IP stack. There are several advantages on running it on top of the Treck stack, such as the full support for the MIB-2 mibs or the capability to run the agent on an IPv6 network if you enable the IPv6 option of Treck.

The agent can however run on top of any TCP/IP stack assuming it supports standard BSD API calls and standard C library functions.

In such a mode, the only MIBs that will be compiled in the agent are the system independent mibs i.e. the system mib, the snmp mib and if SNMPv3 is enabled, the SNMPv3 Framework mibs (VACM, USM, Community, Target and Notification mibs).

To run the agent in such a mode follow these instructions:

### 3.5.1. system settings in trdefs.h:

disable Treck OS:

```
#undefine TM_TRECK_OS, all the code specific to the Treck TCP/IP stacks is under an #ifdef TM_TRECK_OS
```

SNMP version:

Depending of which versions of SNMP you purchased and which version you wish to run, uncomment either one of those:

```
/* #define TM_SNMP_VERSION 1 */
```

```
/* #define TM_SNMP_VERSION 2 */
```

```
/* #define TM_SNMP_VERSION 3 */
```

#### Processor Macros:

Indicate your processor byte ordering, uncomment one of the following macros:

```
/* #define TM_LITTLE_ENDIAN */  
/* #define TM_BIG_ENDIAN */
```

#### Socket API includes:

You may also want to add the correct `#include` to the socket API for your system. (e.g. for windows, `#include <winsock.h>`, for unix, `#include <sys/socket.h>` `#include <netinet/in.h>`, etc...)

#### Socket API and C Library routines:

The agent uses macros for C library and BSD socket API calls starting with “tm\_snmp” e.g. `tm_snmp_strerror`, `tm_snmp_recvfrom`, etc... When `TM_TRECK_OS` is not defined they are defined to use the standard CLIB and BSD API ones (`strncpy`, `recvfrom`, etc...) so it should work by default but you may have to change those definitions to adapt the agent to your system.

### 3.5.2. System-dependent functions required by SNMP:

`tt32Bit tfGetMibUptime(void)`

This function must return the time elapsed (in 100th of seconds) since the agent started. The value returned by this function is used for traps (timestamp field of v1 trap PDU) and is returned to an SNMP manager requesting the `system.sysUpTime.0` object. If you have enabled SNMPv3 it is also used for the replay protection mechanism included in SNMPv3.

`tt32Bit tfGetRouterId(void)`

This function must return a 4-byte unique identifier (usually the agent IPv4 address). It is also needed for v1 traps.

### 3.6 Testing

This agent can be tested with any SNMP compliant manager package, such as HP Openview, Cabletron Spectrum, Tivoli, or others.

The least expensive (though fully compliant) packages are:

Castlerock SNMPc, <http://www.castlerock.com>

MG-SOFT MIB Browser, <http://www.mg-soft.com>

UCD SNMP Project (free), <http://net-snmp.sourceforge.net/snmpv3.html>

SilverCreek from IW Labs, <http://www.iwl.com>

Generally, to test the agent, the user can start the agent and a manager, and then execute SNMP commands on the manager side which exercise the agent. For example, to test snmpwalk of the MIB-II system group, execute the following command using the UCD SNMP manager, where “192.168.100.9” is replaced by the IP address of the SNMP agent and “public” is replaced by a community string valid for retrieval:

```
snmpwalk 192.168.100.9 public .1.3.6.1.2.1.1
```

The output of this command should be something like:

```
system.sysDescr.0 = Treck SNMP Agent
system.sysObjectID.0 = OID: enterprises.29999.1
system.sysUpTime.0 = Timeticks: (1036700) 2:52:47.00
system.sysContact.0 = Treck HQ
system.sysName.0 = Treck, Inc.
system.sysLocation.0 = San Francisco, CA USofA
system.sysServices.0 = 72
```

## 3.7 MIB Compiler and Code Generator Procedures

### 3.7.1 MIB Compiler

The main purpose for any MIB compiler is to verify the syntax and validity of the MIB. MIBs are written in accordance with the ASN.1 rules (somewhat, it's rather a subset of ASN.1) and involve very strong typing, as is the case with all formal languages. A developer writing a new MIB would be well advised to run it through the parser and eliminate all improper constructs, before using or publishing it. All public MIBs undergo such testing before being published.

Treck SNMP MIB compiler is based on `libsmi` (<http://www.ibr.cs.tu-bs.de/projects/sming/>). `libsmi` is an industry standard distribution widely used and recognized. It was developed by SNMP and SMI experts. The core of the `libsmi` distribution is a library that allows management applications to access SMI MIB module definitions. On top of this library, there are tools to check, analyze, dump, convert, and compare MIB definitions. The Treck SNMP mib compiler has been integrated to one of these tools: `smidump`.

### 3.7.2 Installing the mib compiler:

MS WINDOWS:

1. Run the self-extractable zip file. The `libsmi` package is extracted in a new subdirectory called `smi`.
2. Move this directory (including the sub directory) where it suits you e.g. `c:\` or `c:\program files`.
3. Add the `smi\bin` sub directory to your `PATH`.
4. Make the environment variable `SMIPATH` point to the `smi\mibs\ietf` directory. If you have installed the `smi` directory in `c:\` you need not change the `SMIPATH` environment variable.

Note: The MS windows version of the mib compiler (`smidump`) and other tools included in the package are standalone WIN32 executable. They do not require any DLLs.

UNIX:

From the binary:

Linux (x86) binaries are available. To avoid conflicts with existing `libsmi` and facilitate updates, these binaries are statically linked.

1. Uncompress the `tar.gz` file (`tar xzf mibcomp.tar.gz`). The `smi` directory is created and files are extracted in this directory.
2. The tools (including `smidump`, the mib compiler) are located in `smi/bin`. Copy them in a directory that suits you.
3. `mibs` are located in `smi/mibs`. Move this directory where it suits you. By default, `smidump` looks for MIB modules in `/usr/local/share/mibs/ietf`. You can override this option by defining the environment variable `SMIPATH`.

### 3.7.3 Generating the code with the mib compiler

Before, you generate the code for a new MIB, you should make sure that it exists in the `mibs/ietf` sub directory created during `libsmi` installation. There are a wide range of mib files in this directory, unless you want to generate the code for a proprietary mib, it is likely it is already in this directory. If it is not, copy it in this directory as well as any other mib it requires (for imports, textual conventions, etc...) The file must be named according to the mib module name. The mib module name is the one that comes before the "DEFINITIONS ::= BEGIN" statement in the mib definition. For instance, in RFC 1628 (UPS Management Information Base), you can see that the mib definition starts with `UPS-MIB DEFINITIONS ::= BEGIN` therefore the file containing the UPS mib definition has been named `UPS-MIB`.

Once your mib is correctly installed in the `mibs/ietf` subdirectory, you can generate the code using `smidump`. `smidump` supports numerous output formats. To generate the Treck SNMP agent support files and stubs, use the `-f Treck` option:

```
smidump -f Treck [Treck-options] MIBNAME
```

The following options are supported:

<code>--Treck-no-varh</code>	do not generate the var.h file.
<code>--Treck-no-varc</code>	do not generate the var.c file
<code>--Treck-no-localh</code>	do not generate the local.h file
<code>--Treck-no-localc</code>	do not generate the local.c file
<code>--Treck-no-ins</code>	do not generate the .ins file
<code>--Treck-overwrite</code>	overwrite existing files. Use with care since you may lose existing code you have already added in the stub file (local.c file).

Example:

```
C:\smidump -f Treck UPS-MIB
File ups-mib_var.h: DONE
File ups-mib_local.h: DONE
File ups-mib.ins: DONE
File ups-mib_local.c: DONE
File ups-mib_var.c: DONE
```

Warnings:

In many cases, smidump will generate warnings when parsing the MIB file. Typical warnings are range or size restriction missing for objects in the MIB. While it is ok to ignore most of them, it is preferable to correct the mib file from all the errors that it may contain and regenerate the code.

### 3.7.4 Hooking the generated code in the SNMP agent:

The mib compiler generates 5 files per mib. The file names are prefixed with the MIB module name (lowercased) and end with one of the following suffix: `_local.c`, `_local.h`, `_var.c`, `_var.h`, `.ins`.

“local.c” file:

This is the stub file. It contains the “get” stub routines (called when an SNMP GET or GETNEXT is performed on an object of the mib) and the “write” stub routines (called when an SNMP SET is performed). There is one get function per table or group of scalar objects. There is one SET function per table or group of scalar objects only if there are writeable objects in the table or the group of scalar. The get and write routines should be implemented by the user.

“local.h” file:

This file contains C structure definitions that are used by the stub functions. There is one structure per table and one structure per group of scalar object. This file should not be modified.

“var.c” file:

This file interfaces the stub routines to the SNMP agent. This file should not be modified by the user.

“var.h” file:

Various C definitions used to interface the SNMP agent with the user code (in the local.c file). This file should not be modified by the user.

“.ins” file:

This file contains generated code that should be inserted in specific files of the SNMP agent so that the new MIB is completely “hooked” to the agent. Follow the instructions contained in this file.

### 3.8 Configuration APIs

The proper functioning of a SNMP agent depends on a number of parameters not explicitly specified by SNMP protocol, such as trap sink IP address, or a number of communities. The configuration API functions are provided so the user can properly initialize the agent, as well as to modify some parameters during the runtime. Some SNMP objects bear a 'read-only' clause, and, therefore, can not be modified via SNMP operations, but since they reflect particular system properties, the user should be able to modify them. They include sysDescr, sysObjectID and sysServices. The functions are provided for the modification of these objects. The other MIB-II object instances with the clause of 'read-write' can be modified via SNMP operations. No configuration API functions are provided for a modification of these objects.

---

***NOTE: the functions `tfModifyAuthPassword`, `tfModifyPrivacyPassword` and `tfSetSnmPort` may be used ONLY prior to starting the agent. See functions descriptions for full information.***

---

### 3.9 Protocol Stack-Specific functions

These functions are used to retrieve and modify the MIB-II object instances for the following groups: interfaces, at, ip, icmp, tcp and udp. As such, they are highly dependent on the protocol stack particular implementation. The function prototypes, however, are not OS and protocol stack dependent. The function implementations in this distribution conform to the Treck protocol stack implementation. If a developer desires to use other operating system and/or other protocol stack, he/she must remove the implementations of these functions and insert his/her own code allowing a retrieval and modification of MIB-II object instances under a new protocol stack implementation. The function prototypes should not be changed.

## 3.10 Contents of the Distribution media

The entire distribution consists of 10 source files and 16 header files.

### 3.10.1 Source Files

tragent.c. High-level general parsing and building functions  
trapi.c. Configuration API functions.  
trdes.c. DES encryption used by SNMPv3 privacy option.  
trll.c. Low level ASN.1 and BER parsing and building functions.  
trlocal.c. Protocol stack dependent MIB-II objects retrieval and modification function  
trm2vars.c. Protocol stack independent MIB-II objects retrieval and modification  
trmain.c. Main 'select' loop function, trap functions  
trv2c\_v3.c. Functions specific to SNMPv2c and SNMPv3.  
trv3init.c. SNMPv3 initialization function.  
trv3util.c. Miscellaneous SNMPv3 utility functions.  
trv3vars.c. SnmpV2 branch retrieval and modification functions.  
trvars.c. Variable tree searching and comparison functions.

### 3.10.2 Header Files

trapi.h. The primary header file that the user needs to include in their application.  
trasn1.h. ASN.1 and BER definitions, low level functions prototypes.  
trcfg.h. Default values of configurable parameters, used for initialization.  
trconcept.h. Conceptual row tables definitions and struct.  
trconsts.h. MIB-II 'magic number' definitions and trap structs.  
trdefs.h. Treck-specific definitions, translations of agent functions to Treck functions.  
trdeskey.h. DES key definitions.  
trhmacev.h. MD5 and SHA-1 structs and definitions.  
trimpl.h. Implementation-dependent PDU structs and definitions.  
trintapi.h. Function prototypes for internal APIs.  
trmib.h. MIB-II structs and SnmpV2 definitions.  
trmib2.h. typedef's used by MIB-II retrieval functions.  
trregstr.h. Configuration definitions, global struct.  
trsnmp.h. SNMP versions, PDUs, errors and traps definitions.  
trsnmpv3.h. SnmpV2 structs and definitions.  
trv3def.h. SNMP version definitions.  
trvars.h. Variable tree structs.

## 4 Programmer's Reference

The functions in the following sections are classified into the following categories:

- **Core Agent Functions**

These functions represent the main part of the agent. They are SNMP version independent. They are also independent of a particular operating system and a protocol stack.

- **Configuration API Functions**

These functions allow the user to configure the agent. They are independent of SNMP version, an operating system and a protocol stack.

- **Protocol Stack-Specific Functions**

These functions perform retrieval and setting operations on MIB-II objects. They are highly dependent on the operating system and protocol stack.

## 4.1 Core agent functions

The user needs to #include the header file trapi.h in their code which calls these APIs.

### 4.1.1 tfSnmpdMain

```
int          tfSnmpdMain
(
int          blockingState,
unsigned long timerInterval
);
```

#### Function Description

This function is called to initialize and start the SNMP Agent. The SNMP Agent may be run in either blocking or non-blocking mode, specified using the blockingState parameter. If non-blocking mode is selected (i.e. blockingState = TM\_BLOCKING\_OFF), then tfSnmpdMain initializes the agent and returns, and all processing of received SNMP messages occurs later in the context of a socket callback function. If blocking mode is selected (i.e. blockingState = TM\_BLOCKING\_ON), tfSnmpdMain does not return, but instead loops making a blocking select call on the SNMP socket waiting for SNMP messages to be received, and processes them in the context of tfSnmpdMain. Choose blocking mode if you are using a RTOS/Kernel and prefer to run the SNMP Agent as a separate task. Choose non-blocking mode if you do not have a RTOS/Kernel.

#### Parameters

Parameter	Description
<i>blockingState</i>	The flag taking on values TM_BLOCKING_OFF or TM_BLOCKING_ON
<i>timerInterval</i>	Not used

#### Returns

Value	Meaning
0	Success.
1	Failure to allocate memory for the global struct tvSnmpdPtr.
2	Failure to initialize the agent

### 4.1.2 tfSnmpdStop

```
int          tfSnmpdStop
(
void
);
```

#### Function Description

This function is called to close the SNMP sockets and kill the SNMP Agent. tfSnmpdStop does the reverse of tfSnmpdMain, i.e. any memory that was allocated for the SNMP Agent during initialization is now freed.

#### Parameters

None

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success

## 4.2 Non-Runtime Initialization functions

The following initialization APIs may only be called prior to starting the SNMP Agent (i.e. prior to calling `tfSnmpdMain`). The user must `#include` the header file `trapi.h` in their code which calls these APIs.

### 4.2.1 `tfModifyAuthPassword`

```
int          tfModifyAuthPassword
(
char        *password
);
```

#### Function Description

This function lets the user modify the authentication password. Maximum length of `password` human-readable string is set as `TM_SNMP_MAX_USER_AUTH_PSWD_LEN`. It is a responsibility of a calling function to allocate memory for `password`. The user can call this function only prior to starting the agent to overwrite the default.

#### Parameters

Parameter	Description
<i>password</i>	Input parameter, password string to be modified

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables
<code>TM_AGENT_RUNNING</code>	Failure, agent is already running.
<code>TM_PASSWORD_TOOLONG</code>	Failure, password is too long
<code>TM_PASSWORD_NOT_PRINTABLE</code>	Failure, password contains unprintable characters

---

### 4.2.2 tfModifyPrivacyPassword

```
int          tfModifyPrivacyPassword
(
char         *password
);
```

#### Function Description

This function lets the user modify the privacy password. Maximum length of `password` human-readable string is set as `TM_SNMP_MAX_USER_PRIVACY_PSWD_LEN`. It is a responsibility of a calling function to allocate memory for `password`. The user can call this function only prior to starting the agent to overwrite the default.

#### Parameters

Parameter	Description
<i>password</i>	Input parameter, password string to be modified

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables
<code>TM_AGENT_RUNNING</code>	Failure, agent is already running
<code>TM_PASSWORD_TOOLONG</code>	Failure, password is too long
<code>TM_PASSWORD_NOT_PRINTABLE</code>	Failure, password contains unprintable characters

### 4.2.3 `tfSetSnmPort`

```
int          tfSetSnmPort
(
int          portNumber
);
```

#### Function Description

This function lets the user set the port number on which SNMP agent receives SNMP requests to `portNumber`. The port is defined in `trconsts.h` as `snmpPort` inside `ttSnmPd`. By default, when the agent starts, if `snmpPort` is zero, the agent initializes it with the default value `SNMP_PORT` defined in `trcfg.h`. The user can call this function only prior to starting the agent to overwrite the default.

#### Parameters

Parameter	Description
<i>portNumber</i>	Port numeric value on input.

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_AGENT_RUNNING</code>	Failure, agent is already running.
<code>TM_PORT_TOOLARGE</code>	Failure, <i>portNumber</i> value is larger than 65535.

#### 4.2.4 tfModifySysDescr

```
int          tfModifySysDescr
(
char        *sysDescr
);
```

##### Function Description

This function lets the user modify MIB-II sysDescr object instance, which cannot be changed via SNMP SET request due to its ACCESS clause of 'read-only'. By definition, sysDescr is "A textual description of the entity. This value should include the full name and version identification of the system's hardware type, software operating-system, and networking software. It is mandatory that this only contain printable ASCII characters. ". Its length must not exceed 255 bytes. The agent initializes it to the default value of the octet string tvSnmpSysDescription, defined in trcfg.h. This function allows the user to change this default value. It is a responsibility of a calling function to allocate memory for sysDescr. The user must call this function prior to starting the agent, since sysDescr object instance must remain unchanged during the agent's lifetime.

##### Parameters

Parameter	Description
<i>sysDescr</i>	Input parameter, an octet string value of sysDescr MIB-II object

##### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.
TM_SYSDESCR_TOOLONG	Failure, sysDescr contains too many characters.
TM_SYSDESCR_NOT_PRINTABLE	Failure, sysDescr contains unprintable characters.

TM\_SYSLOCATION\_NOT\_PRINTABLE Failure, sysLocation contains unprintable characters.

## 4.2.5 tfModifySysObjid

```
int          tfModifySysObjid
(
unsigned long *sysOid,
int          oidLen
);
```

### Function Description

This function lets the user modify MIB-II sysObjectID object instance, which cannot be changed via SNMP SET request due to its ACCESS clause of 'read-only'. By definition, sysObjectID is "The vendor's authoritative identification of the network management subsystem contained in the entity. This value is allocated within the SMI enterprises subtree (1.3.6.1.4.1) and provides an easy and unambiguous means for determining 'what kind of box' is being managed. For example, if vendor 'Flintstones, Inc.' was assigned the subtree 1.3.6.1.4.1.4242, it could assign the identifier 1.3.6.1.4.1.4242.1.1 to its 'Fred Router'.". Its length must not exceed TM\_SNMP\_MAX\_NAME\_LEN defined in trimpl.h. The agent initializes it to the default value of the octet string tvSnmpSysObjid, defined in trcfg.h. This function allows the user to change this default value. It is a responsibility of a calling function to allocate memory for tvSnmpSysObjid. The user must call this function prior to starting the agent, since tvSnmpSysObjid object instance must remain unchanged during the agent's lifetime.

### Parameters

Parameter	Description
<i>sysOid</i>	Input parameter, a pointer to a new sysObjectID MIB-II object value
<i>oidLen</i>	Input parameter, length (in unsigned long units) of <i>sysOid</i> .

### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success.
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.
TM_SYSOID_TOOLONG	Failure, sysOid is too long.

## 4.2.6 tfModifySysServices

```
int          tfModifySysServices
(
int          sysServices
);
```

### Function Description

This function lets the user modify MIB-II sysServices object instance, which cannot be changed via SNMP SET request due to its ACCESS clause of 'read-only'. By definition, sysServices is "A value which indicates the set of services that this entity primarily offers. The value is a sum. This sum initially takes the value zero, Then, for each layer, L, in the range 1 through 7, that this node performs transactions for, 2 raised to (L - 1) is added to the sum. For example, a node which performs primarily routing functions would have a value of 4 ( $2^{(3-1)}$ ). In contrast, a node which is a host offering application services would have a value of 72 ( $2^{(4-1)} + 2^{(7-1)}$ ). Note that in the context of the Internet suite of protocols, values should be calculated accordingly:

layer	functionality
1	physical (e.g., repeaters)
2	datalink/subnetwork (e.g., bridges)
3	internet (e.g., IP gateways)
4	end-to-end (e.g., IP hosts)
7	applications (e.g., mail relays)

For systems including OSI protocols, layers 5 and 6 may also be counted." The agent initializes it to the default value of `tvSnmpSysServices`, defined in `trcfg.h`. This function allows the user to change this default value. The user must call this function prior to starting the agent, since `tvSnmpSysServices` object instance must remain unchanged during the agent's lifetime.

### Parameters

Parameter	Description
<code>sysServices</code>	Input parameter, an integer value of sysServices MIB-II object

### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_SYSSERVICES_TOOLARGE</code>	Failure, sysServices larger than 127

#### 4.2.7 `tfSetColdStartTrap`

```
int tfSetColdStartTrap
(
  const char *comName,
  const struct sockaddr_storage *trapDestPtr,
  int flags
);
```

#### Function Description

This function lets the user configure the Cold start trap the agent will send. When the agent starts, it can send cold start traps to an SNMP manager to indicate it is up and running. This function must be called prior to starting the agent, since cold start traps are sent only at startup.

#### Parameters

Parameter	Description
<i>comName</i>	Community name that will be embedded in the trap.
<i>trapDestPtr</i>	A pointer to a <code>sockaddr_storage</code> structure that indicates the destination address of the trap. The <code>addr</code> , <code>ss_len</code> , <code>ss_family</code> and <code>ss_port</code> fields of the <code>sockaddr_storage</code> structure must be correctly set.
<i>flags</i>	By default, one SNMP V1 trap, one V2 trap (if the agent is running SNMPv2c) and one V3 traps (if the agent is running SNMPv3) are sent (one cold start trap per SNMP version the agent supports). You can use the flags variable to avoid sending multiple cold start traps to the manager. The first bit of the flags variable is reserved, you can set the 2 <sup>nd</sup> , 3 <sup>rd</sup> and 4 <sup>th</sup> bits to prevent the agent from sending a V1, V2c or V3 coldStart trap. To use the default behavior, set the <i>flags</i> to 0, otherwise the following macros should be used to compute the mask. TM_SNMP_CLDSTRT_TRAP_NO_V1 TM_SNMP_CLDSTRT_TRAP_NO_V2C TM_SNMP_CLDSTRT_TRAP_NO_V3

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables
TM_AGENT_RUNNING	Failure, trap sinks list is locked
TM_COMMUNITY_TOOLONG	Failure, comName is too long
TM_COMMUNITY_NOT_PRINTABLE	Failure, comName contains unprintable characters

### 4.3 Configuration API functions

Some configuration API functions may only be called prior to starting the SNMP Agent (i.e. prior to calling `tfSnmPdMain`), and these are listed separately in the previous section as “*non-runtime initialization*” functions.

The other configuration APIs listed in this section may be called anytime.

The user must `#include` the header file `trapi.h` in their code which calls these APIs.

#### 4.3.1 `tfAddCommunity`

```
int          tfAddCommunity
(
int          which,
char        *community
);
```

#### Function Description

This function lets the user add a community string to the list of communities. Two pools of communities, the sizes of `TM_SNMP_MAX_NUM_READ_COMMUNITIES` and `TM_SNMP_MAX_NUM_WRITE_COMMUNITIES`, are defined. By default, when the agent starts, if either community pool is empty, the agent initializes with one default READ community and one default WRITE community. They might or might not be identical. Maximum length of the community string in each pool is set as `TM_SNMP_MAX_COMMUNITY_LEN`.

It is the responsibility of a calling function to allocate memory for `community`. The user can call this function any time prior to starting the agent, or during the runtime. In trilingual mode (`TM_SNMP_VERSION` *#defined* to 3), the communities are handled in the *community table* which is part of the SNMPv3 framework. In trilingual mode, use one of the community table APIs (`tfAddCommunityTableEntry`, `tfDeleteCommunityTableEntry`) instead of this API.

#### Parameters

Parameter	Description
<i>which</i>	Input parameter, an integer value taking on <code>TM_READ_COMMUNITY_FLAG</code> , or <code>TM_WRITE_COMMUNITY_FLAG</code>
<i>community</i>	Input parameter, community string value

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success
	<code>TM_ALLOCATION_ERROR</code>
	Failure to allocate memory for the agent global variables.
<code>TM_RECORD_LOCKED</code>	Failure, communities list is locked
<code>TM_NOMORE_COMMUNITIES</code>	Failure, communities list is full
<code>TM_UNKNOWN_COMMUNITY_FLAG</code>	Failure, which value is not defined
<code>TM_COMMUNITY_TOOLONG</code>	Failure, community is too long
<code>TM_COMMUNITY_NOT_PRINTABLE</code>	Failure, community contains unprintable characters

### 4.3.2 `tfAddTrapEntry`

---

*`tfAddTrapEntry` has been deprecated.  
Please use `Use tfNgAddTrapEntry` instead.*

---

### 4.3.3 `tfDeleteAccessEntry`

```
int                tfDeleteAccessEntry
(
ttSnmPVacmAccessTable *x
);
```

#### Function Description

This function deletes the entry from the linked list of `ttSnmPVacmAccessTable`'s. The struct `ttSnmPVacmAccessTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPVacmAccessTable
{
    struct tsSnmPVacmAccessTable *svatNext;
    int svatGroupNameLen;
    int svatPrefixLen;
    int svatModel;
    int svatLevel;
    int svatMatch;
    int svatReadNameLen;
    int svatWriteNameLen;
    int svatNotifyNameLen;
    int svatStorageType;
    int svatStatus;
    char svatGroupName[32];
    char svatPrefix[32];
    char svatReadName[32];
    char svatWriteName[32];
    char svatNotifyName[32];
} ttSnmPVacmAccessTable;
```

The table index consists of `svatGroupName`, `svatPrefix`, `svatModel` and `svatLevel` (in that order). Note that the user must specify all index fields (together with their length values) only. It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<code>x</code>	Input parameter, pointer to <code>ttSnmPVacmAccessTable</code> , with first six fields filled in.

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_WRONG_GROUPNAME_LENGTH</code>	Failure, <code>svatGroupNameLen</code> is not between 1 and 32.
<code>TM_GROUPNAME_NOT_PRINTABLE</code>	Failure, <code>svatGroupName</code> contains unprintable characters.
<code>TM_WRONG_PREFIX_LENGTH</code>	Failure, <code>svatPrefixLen</code> is not between 1 and 32.
<code>TM_PREFIX_NOT_PRINTABLE</code>	Failure, <code>svatPrefix</code> contains unprintable characters.
<code>TM_WRONG_SECURITYMODEL</code>	Failure, <code>svatModel</code> < 0.
<code>TM_WRONG_SECURITYLEVEL</code>	Failure, <code>svatLevel</code> is not 1, or 2, or 3
<code>TM_WRONG_MATCH</code>	Failure, <code>svatMatch</code> is not 0, or 1, or 2
<code>TM_WRONG_STATUS</code>	Failure, <code>svatStatus</code> is not 1, or 2, or 3
<code>TM_RECORD_NOTFOUND</code>	Failure to find the indicated entry in the linked list

### 4.3.4 `tfAddCommunityTableEntry`

```
int tfAddCommunityTableEntry
(
tt32Bit flags,
struct tsSnmCommunityTable *comEntryPtr)
);
```

#### Function Description

This function lets the user add an entry in the community table. The community table (RFC 2576) is part of the SNMPv3 framework and provides a way to define a v1/v2c community and map it to a SNMPv3 USM user. The same access rights that apply to the V3 user apply to the community.

The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

##### Parameter Description

*flags*: input parameter - unused

*comEntryPtr*: Input parameter, pointer to a struct `tsSnmCommunityTable` that the user must allocate and fill.

```
typedef struct tsSnmCommunityTable
{
    struct tsSnmCommunityTable *next;
    tt8Bit    snmpCommunityIndex[TM_SNMP_COMTABLE_MAX_NAME_LEN];
    tt8Bit    snmpCommunityName[TM_SNMP_COMTABLE_MAX_NAME_LEN];
    tt8Bit    snmpCommunitySecurityName[TM_SNMP_COMTABLE_MAX_NAME_LEN];
    tt8Bit    snmpCommunityContextEngineID[TM_SNMP_COMTABLE_MAX_NAME_LEN];
    tt8Bit    snmpCommunityContextName[TM_SNMP_COMTABLE_MAX_NAME_LEN];
    tt8Bit    snmpCommunityTransportTag[TM_SNMP_COMTABLE_MAX_NAME_LEN];
    tt8Bit    snmpCommunityIndexLen;
    tt8Bit    snmpCommunityNameLen;
    tt8Bit    snmpCommunitySecurityNameLen;
    tt8Bit    snmpCommunityContextEngineIDLen;
    tt8Bit    snmpCommunityContextNameLen;
    tt8Bit    snmpCommunityTransportTagLen;
    tt8Bit    snmpCommunityStorageType;
    tt8Bit    snmpCommunityStatus;
} ttSnmCommunityTable;
```

`snmpCommunityIndex`, `snmpCommunityIndexLen`:  
community identifier (character string)

`snmpCommunityName`, `snmpCommunityNameLen`:  
The community string (e.g. "public", "private", etc...)

`snmpCommunitySecurityName`, `snmpCommunitySecurityNameLen`,  
`snmpCommunityContextEngineID`, `snmpCommunityContextEngineIDLen`,  
`snmpCommunityContextName`, `snmpCommunityContextNameLen`:

Parameters which identify a row of the `usmUserTable`.

`snmpCommunityTransportTag`, `snmpCommunityTransportTagLen`:  
Unused parameters, defined for future use.

`snmpCommunityStorageType`:  
The table row storage type. Use one of the following macros to indicate the storage type of the row:

- `TM_SNMP_STORAGETYPE_OTHER`
- `TM_SNMP_STORAGETYPE_VOLATILE`
- `TM_SNMP_STORAGETYPE_NONVOLATILE`

*Note: TM\_SNMP\_STORAGETYPE\_NONVOLATILE should be used for RAM storage,*

*TM\_SNMP\_STORAGETYPE\_NONVOLATILE for ROM storage.*

*The value chosen is just an indication and does not affect the behavior of the agent.*

---

snmpCommunityStatus:

The table row status. Use one of the three allowed macros to indicate the row status:

- TM\_SNMP\_ROWSTATUS\_ACTIVE
- TM\_SNMP\_ROWSTATUS\_NOTINSERVICE
- TM\_SNMP\_ROWSTATUS\_NOTREADY

TM\_SNMP\_ROWSTATUS\_ACTIVE should be used with the API, the other values are used for conceptual row creation via SNMP SET messages.

## Returns

### Value

TM\_SNMPAPI\_NOERROR  
TM\_COMMUNITY\_NOT\_PRINTABLE  
TM\_USER\_NOT\_PRINTABLE:  
TM\_DUPLICATE\_COMMUNITY:

### Meaning

Success  
Failure, community contains unprintable characters.  
Failure, user name contains unprintable characters.  
Failure, the community already exists.

### 4.3.5 tfDeleteCommunity

```
int          tfDeleteCommunity
(
int          which,
char        *community
);
```

#### Function Description

This function lets the user delete a community string from the list of communities. Two pools of communities, the sizes of `TM_SNMP_MAX_NUM_READ_COMMUNITIES` and `TM_SNMP_MAX_NUM_WRITE_COMMUNITIES`, are defined. By default, when the agent starts, if either community pool is empty, the agent initializes with one default READ and one default WRITE communities. They might or might not be identical. Maximum length of the community string in each pool is set as `TM_SNMP_MAX_COMMUNITY_LEN`. It is a responsibility of a calling function to allocate memory for community. If this community is the only one left in the pool, it can not be deleted. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>which</i>	Input parameter, an integer value taking on <code>TM_READ_COMMUNITY_FLAG</code> , or <code>TM_WRITE_COMMUNITY_FLAG</code>
<i>community</i>	Input parameter, community string value

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_RECORD_LOCKED</code>	Failure, communities list is locked.
<code>TM_RECORD_NOTFOUND</code>	Failure, community is not found.
<code>TM_UNKNOWN_COMMUNITY_FLAG</code>	Failure, which value is not defined.
<code>TM_COMMUNITY_TOOLONG</code>	Failure, community is too long.
<code>TM_COMMUNITY_NOT_PRINTABLE</code>	Failure, community contains unprintable characters.
<code>TM_DEFAULT_NOTDELETED</code>	Failure, the last community cannot be deleted.

### 4.3.6 `tfDeleteCommunityTableEntry`

```
int tfDeleteCommunityTableEntry  
(  
    tt32Bit flags,  
    char *comIndex  
);
```

#### Function Description

This function lets the user remove a community from the community table. The community table (RFC 2576) is part of the SNMPv3 framework and provides a way to define a v1/v2c community and map it to a SNMPv3 USM user. The same access rights that apply to the V3 user apply to the community.

The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>flags:</i>	input parameter - unused
<i>comIndex:</i>	Input parameter, index of the community to remove.

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_RECORD_NOTFOUND	Failure, the community could not be found.

### 4.3.7 tfDeleteContextEntry

```
int                tfDeleteContextEntry
(
ttSnmPVacmContextTable *x
);
```

#### Function Description

This function deletes the entry from the linked list of `ttSnmPVacmContextTable`'s. The struct `ttSnmPVacmContextTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPVacmContextTable
{
    struct tsSnmPVacmContextTable *svctNext;
    int svctNameLen;
    char svctName[32];
} ttSnmPVacmContextTable;
```

The table index consists of `svctName`. Note that the user must specify index field (together with `svctNameLen`). It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<code>x</code>	Input parameter, pointer to <code>ttSnmPVacmContextTable</code> , with <code>svctName</code> and <code>svctNameLen</code> filled in.

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_WRONG_CONTEXTNAME_LENGTH</code>	Failure, <code>svctNameLen</code> is not between 0 and 32.
<code>TM_CONTEXTNAME_NOT_PRINTABLE</code>	Failure, <code>svctName</code> contains unprintable characters.
<code>TM_RECORD_NOTFOUND</code>	Failure to find the indicated entry in the linked list.

### 4.3.8 tfDeleteSec2GroupEntry

```
int                tfDeleteSec2GroupEntry
(
ttSnmPVacmSec2GroupTable  *x
);
```

#### Function Description

This function deletes the entry from the linked list of `ttSnmPVacmSec2GroupTable`'s. The struct `ttSnmPVacmAccessTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPVacmSec2GroupTable
{
    struct tsSnmPVacmSec2GroupTable *svgtNext;
    int svgtModel;
    int svgtNameLen;
    int svgtGroupNameLen;
    int svgtStorageType;
    int svgtStatus;
    char svgtName[32];
    char svgtGroupName[32];
} ttSnmPVacmSec2GroupTable;
```

The table index consists of `svgtModel`, and `svgtName` (in that order). Note that the user must specify both index fields (together with `svgtNameLen`). It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

##### Parameter

`x`

##### Description

Input parameter, pointer to `ttSnmPVacmSec2GroupTable`, with first two fields, and `svgtGroupName`, `svgtGroupNameLen` filled in.

#### Returns

##### Value

`TM_SNMPAPI_NOERROR`

`TM_ALLOCATION_ERROR`

`TM_WRONG_SECURITYMODEL`

`TM_WRONG_SECURITYNAME_LENGTH`

`TM_SECURITYNAME_NOT_PRINTABLE`

`TM_WRONG_GROUPNAME_LENGTH`

`TM_GROUPNAME_NOT_PRINTABLE`

`TM_WRONG_STATUS`

`TM_RECORD_NOTFOUND`

##### Meaning

Success.

Failure to allocate memory for the agent global variables.

Failure, `svgtModel < 1`.

Failure, `svgtNameLen` is not between 1 and 32.

Failure, `svgtName` contains unprintable characters.

Failure, `svgtGroupNameLen` is not between 1 and 32.

Failure, `svgtGroupName` contains unprintable characters.

Failure, `svgtStatus` is not 1, or 2, or 3.

Failure to find the indicated entry in the linked list.

### 4.3.9 `tfDeleteTrapEntry`

---

*`tfDeleteTrapEntry` has been deprecated.  
Please use `tfNgDeleteTrapEntry` instead.*

---

### 4.3.10 tfDeleteUsmEntry

```
int                tfDeleteUsmEntry
(
ttSnmPUsmUserTable *x
);
```

#### Function Description

This function deletes the entry from the linked list of `ttSnmPUsmUserTable`'s. The struct `ttSnmPUsmUserTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPUsmUserTable
{
    struct tsSnmPUsmUserTable *suutNext;
    oid suutCloneFrom[TM_SNMP_MAX_NAME_LEN];
    oid suutAuthProtocol[TM_SNMP_MAX_NAME_LEN];
    oid suutPrivProtocol[TM_SNMP_MAX_NAME_LEN];
    int suutSecurityNameLen;
    int suutCloneFromLen;
    int suutAuthProtocolLen;
    int suutAuthKeyChangeLen;
    int suutOwnAuthKeyChangeLen;
    int suutPrivProtocolLen;
    int suutPrivKeyChangeLen;
    int suutOwnPrivKeyChangeLen;
    int suutPublicLen;
    int suutStorageType;
    int suutStatus;
    char suutSecurityName[32];
    char suutAuthKeyChange[32];
    char suutOwnAuthKeyChange[32];
    char suutPrivKeyChange[32];
    char suutOwnPrivKeyChange[32];
    char suutUuPublic[32];
    char suutName[32];
    unsigned char suutEngineId[20];
    unsigned char suutEngineIdLen;
    unsigned char suutNameLen;
    char suutPad[2];
} ttSnmPUsmUserTable;
```

The table index consists of `suutEngineId` and `suutName` (in that order). Note that the user must specify both index fields (together with their length values). It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

**Parameters****Parameter***x***Description**

Input parameter, pointer to ttSnmPUsmUserTable, with first four fields filled in.

**Returns****Value**

TM\_SNMPAPI\_NOERROR

**Meaning**

Success

TM\_ALLOCATION\_ERROR Failure to allocate memory for the agent global variables.

TM\_ZERO\_ENGINEID\_LENGTH

Failure, suutEngineIdLen is 0.

TM\_WRONG\_USERNAME\_LENGTH

Failure, suutNameLen is not between 1 and 32.

TM\_USERNAME\_NOT\_PRINTABLE

Failure, suutName contains unprintable characters.

TM\_ZERO\_SECURITYNAME\_LENGTH

Failure, suutSecurityNameLen is 0.

TM\_SECURITYNAME\_NOT\_PRINTABLE Failure, suutSecurityName contains unprintable characters.

TM\_ZERO\_CLONEFROM\_LENGTH

Failure, suutCloneFromLen is 0.

TM\_WRONG\_PUBLIC\_LENGTH

Failure, suutPublicLen is not between 0 and 32.

TM\_PUBLIC\_NOT\_PRINTABLE

Failure, suutPublic contains unprintable characters.

TM\_UNKNOWN\_STORAGE\_TYPE

Failure, suutStorageType is not 0 or between 1 and 5.

TM\_WRONG\_STATUS

Failure, suutStatus is not 1, or 2, or 3.

TM\_RECORD\_NOTFOUND

Failure to find the indicated entry in the linked list.

### 4.3.11 tfDeleteVtfEntry

```
int                                tfDeleteVtfEntry
(
ttSnmPVacmViewTreeFamilyTable    *x
);
```

#### Function Description

This function deletes the entry from the linked list of `ttSnmPVacmViewTreeFamilyTable`'s. The struct `ttSnmPVacmViewTreeFamilyTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPVacmViewTreeFamilyTable
{
    struct tsSnmPVacmViewTreeFamilyTable *svvtNext;
    oid svvtSubtree[TM_SNMP_MAX_NAME_LEN];
    int svvtNameLen;
    int svvtSubtreeLen;
    int svvtMaskLen;
    int svvtType;
    int svvtStorageType;
    int svvtStatus;
    char svvtName[32];
    unsigned char svvtMask[16];
} ttSnmPVacmViewTreeFamilyTable;
```

The table index consists of `svvtName` and `svvtSubtree` (in that order). Note that the user must specify both index fields (together with their length values). It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<code>x</code>	Input parameter, pointer to <code>ttSnmPVacmViewTreeFamilyTable</code> , with first four fields filled in.

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_WRONG_VTFNAME_LENGTH</code>	Failure, <code>svvtNameLen</code> is not between 0 and 32.
<code>TM_VTFNAME_NOT_PRINTABLE</code>	Failure, <code>svvtName</code> contains unprintable characters.
<code>TM_WRONG_TYPE</code>	Failure, <code>svvtType</code> is not 0, or 1, or 2.
<code>TM_WRONG_STATUS</code>	Failure, <code>svvtStatus</code> is not 1, or 2, or 3.
<code>TM_RECORD_NOTFOUND</code>	Failure to find the indicated entry in the linked list.

### 4.3.12 `tfDisplayAccessEntry`

```
int                tfDisplayAccessEntry
(
ttSnmPVacmAccessTable *entryPtr,
unsigned int         i
);
```

#### Function Description

This function lets the user to access the entry (row) from `vacmAccessTable` by returning the pointer to the corresponding struct. It might be convenient to inspect an existing entry prior to deleting one, or adding one, or for any other purpose. The user must specify the entry number, `i` (0-based). It is a responsibility of the calling function to allocate memory to hold the `ttSnmPVacmAccessTable` pointed to by `entryPtr` prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<code>entryPtr</code>	Pointer to memory allocated by the user, which is populated by this function with information copied from the row at the specified index.
<code>i</code>	Index of the row to be displayed

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_RECORD_NOTFOUND</code>	Failure, no such row is not found.

### 4.3.13 `tfDisplayContextEntry`

```
int                tfDisplayContextEntry
(
ttSnmpVacmContextTable *entryPtr,
unsigned int          i
);
```

#### Function Description

This function lets the user to access the entry (row) from `vacmContextTable` by returning the pointer to the corresponding struct. It might be convenient to inspect an existing entry prior to deleting one, or adding one, or for any other purpose. The user must specify the entry number, `i` (0-based). It is a responsibility of the calling function to allocate memory to hold the `ttSnmpVacmContextTable` pointed to by `entryPtr` prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
-----------	-------------

<code>entryPtr</code>	Pointer to memory allocated by the user, which is populated by this function with information copied from the row at the specified index.
-----------------------	---

<code>i</code>	Index of the row to be displayed
----------------	----------------------------------

#### Returns

Value	Meaning
-------	---------

<code>TM_SNMPAPI_NOERROR</code>	Success.
---------------------------------	----------

<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
----------------------------------	--

<code>TM_RECORD_NOTFOUND</code>	Failure, no such row is not found.
---------------------------------	------------------------------------

### 4.3.14 tfDisplaySec2GroupEntry

```

int                tfDisplaySec2GroupEntry
(
ttSnmPVacmSec2GroupTable *entryPtr,
unsigned int         i
);

```

#### Function Description

This function lets the user to access the entry (row) from vacmSecurityToGroupTable by returning the pointer to the corresponding struct. It might be convenient to inspect an existing entry prior to deleting one, or adding one, or for any other purpose. The user must specify the entry number, *i* (0-based). It is a responsibility of the calling function to allocate memory to hold the `ttSnmPVacmSec2GroupTable` pointed to by `entryPtr` prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>entryPtr</i>	Pointer to memory allocated by the user, which is populated by this function with information copied from the row at the specified index.
<i>i</i>	Index of the row to be displayed

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.
TM_RECORD_NOTFOUND	Failure, no such row is not found.

### 4.3.15 tfDisplaySysContact

```
int tfDisplaySysContact
(
char *sysContact
);
```

#### Function Description

This function lets the user to access the sysContact MIB-II object. It is the responsibility of the calling function to allocate memory to hold the sysContact prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>sysContact</i>	Output parameter. Pointer to memory allocated by the user, which is populated by this function with the MIB-II sysContact string.

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.

### 4.3.16 tfDisplaySysDescr

```
int tfDisplaySysDescr
(
char *sysDescr
);
```

#### Function Description

This function lets the user to access the sysDescr MIB-II object. It is the responsibility of the calling function to allocate memory to hold the sysDescr prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>sysDescr</i>	Output parameter. Pointer to memory allocated by the user, which is populated by this function with the MIB-II sysDescr string.

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.

### 4.3.17 tfDisplaySysObjectId

```
int tfDisplaySysObjectId  
(  
char * sysObjectId  
);
```

#### Function Description

This function lets the user to access the sysObjectId MIB-II object. It is the responsibility of the calling function to allocate memory to hold the sysObjectId prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

**Parameter**  
*sysObjectId*

#### Description

Output parameter. Pointer to memory allocated by the user, which is populated by this function with the MIB-II sysObjectId string.

#### Returns

**Value**  
TM\_SNMPAPI\_NOERROR  
TM\_ALLOCATION\_ERROR

#### Meaning

Success  
Failure to allocate memory for the agent global variables.

### 4.3.18 tfDisplaySysObjectId

```
int tfDisplaySysObjectId
(
char *sysObjectId,
int *sysObjectIdLen
);
```

#### Function Description

This function lets the user to access the sysObjectId MIB-II object. It is the responsibility of the calling function to allocate memory to hold the sysObjectId prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

##### Parameter

sysObjectId

sysObjectIdLen

##### Description

Output parameter. Pointer to memory allocated by the user, which is populated by this function with the MIB-II sysObjectId.

Output parameter. Pointer to the number of elements sysObjectId.

#### Returns

##### Value

TM\_SNMPAPI\_NOERROR

TM\_ALLOCATION\_ERROR

##### Meaning

Success

Failure to allocate memory for the agent global variables.

### 4.3.19 tfDisplaySysLocation

```
int tfDisplaySysLocation
(
char *sysLocation
);
```

#### Function Description

This function lets the user to access the sysLocation MIB-II object. It is the responsibility of the calling function to allocate memory to hold the sysLocation prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>sysLocation</i>	Output parameter. Pointer to memory allocated by the user, which is populated by this function with the MIB-II sysLocation string.

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.

### 4.3.20 tfDisplaySysName

```
int tfDisplaySysName
(
char *sysName
);
```

#### Function Description

This function lets the user to access the sysName MIB-II object. It is the responsibility of the calling function to allocate memory to hold the sysName prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>sysName</i>	Output parameter. Pointer to memory allocated by the user, which is populated by this function with the MIB-II sysName string.

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.

#### 4.3.21 `tfDisplayTrapEntry`

---

*`tfDisplayTrapEntry` has been deprecated API.  
Please use `tfNgDisplayTrapEntry` instead.*

---

### 4.3.22 `tfDisplayUsmEntry`

```
int          tfDisplayUsmEntry
(
ttSnmPUsmUserTable *entryPtr,
unsigned int      i
);
```

#### Function Description

This function lets the user to access the entry (row) from `usmUserTable` by returning the pointer to the corresponding struct. It might be convenient to inspect an existing entry prior to deleting one, or adding one, or for any other purpose. The user must specify the entry number, `i` (0-based). It is a responsibility of the calling function to allocate memory to hold the `ttSnmPUsmUserTable` pointed to by `entryPtr` prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>entryPtr</i>	Pointer to memory allocated by the user, which is populated by this function with information copied from the row at the specified index.
<i>i</i>	Index of the row to be displayed

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_RECORD_NOTFOUND</code>	Failure, no such row was found.

### 4.3.23 tfDisplayVtfEntry

```

int                                     tfDisplayVtfEntry
(
ttSnmpVacmViewTreeFamilyTable    *entryPtr,
unsigned int                      i
);

```

#### Function Description

This function lets the user to access the entry (row) from `vacmViewTreeFamilyTable` by returning the pointer to the corresponding struct. It might be convenient to inspect an existing entry prior to deleting one, or adding one, or for any other purpose. The user must specify the entry number, `i` (0-based). It is a responsibility of the calling function to allocate memory to hold the `ttSnmpVacmViewTreeFamilyTable` pointed to by `entryPtr` prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

##### Parameter

*entryPtr*

*i*

##### Description

Pointer to memory allocated by the user, which is populated by this function with information copied from the row at the specified index.

Index of the row to be displayed

#### Returns

##### Value

TM\_SNMPAPI\_NOERROR

TM\_ALLOCATION\_ERROR

TM\_RECORD\_NOTFOUND

##### Meaning

Success.

Failure to allocate memory for the agent global variables.

Failure, no such row was found.

### 4.3.24 `tfInsertAccessEntry`

```
int          tfInsertAccessEntry
(
ttSnmPVacmAccessTable  *x
);
```

#### Function Description

This function either inserts the new entry into the linked list of `ttSnmPVacmAccessTable`'s, if the compound table index does not coincide with any index in the entries in the linked list, or replaces the old entry with the new one, if the identical index is found. The struct `ttSnmPVacmAccessTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPVacmAccessTable
{
    struct tsSnmPVacmAccessTable *svatNext;
    int svatGroupNameLen;
    int svatPrefixLen;
    int svatModel;
    int svatLevel;
    int svatMatch;
    int svatReadNameLen;
    int svatWriteNameLen;
    int svatNotifyNameLen;
    int svatStorageType;
    int svatStatus;
    char svatGroupName[32];
    char svatPrefix[32];
    char svatReadName[32];
    char svatWriteName[32];
    char svatNotifyName[32];
} ttSnmPVacmAccessTable;
```

The table index consists of `svatGroupName`, `svatPrefix`, `svatModel` and `svatLevel` (in that order). Note that the user should specify all index fields (together with their length values) and `svatStatus` value since they do not have DEFVAL clauses. If the user does not specify other fields, they will be assigned their default values, per RFC 2575. It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<code>x</code>	Input parameter, pointer to <code>ttSnmPVacmAccessTable</code> , with the fields <code>svatGroupName</code> , <code>svatGroupNameLen</code> , <code>svatPrefix</code> , <code>svatPrefixLen</code> , <code>svatModel</code> , <code>svatLevel</code> , and <code>svatStatus</code> filled in.

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_WRONG_GROUPNAME_LENGTH</code>	Failure, <code>svatGroupNameLen</code> is not between 1 and 32.
<code>TM_GROUPNAME_NOT_PRINTABLE</code>	Failure, <code>svatGroupName</code> contains unprintable characters.
<code>TM_WRONG_PREFIX_LENGTH</code>	Failure, <code>svatPrefixLen</code> is not between 1 and 32.
<code>TM_PREFIX_NOT_PRINTABLE</code>	Failure, <code>svatPrefix</code> contains unprintable characters.
<code>TM_WRONG_SECURITYMODEL</code>	Failure, <code>svatModel</code> < 0.
<code>TM_WRONG_SECURITYLEVEL</code>	Failure, <code>svatLevel</code> is not 1, or 2, or 3.

TM_WRONG_MATCH	Failure, svatMatch is not 0, or 1, or 2.
TM_WRONG_STATUS	Failure, svatStatus is not 1, or 2, or 3.
TM_ALLOCATION_ERROR	Failure to allocate memory for snmpVacmAccessEntryStart

### 4.3.25 tfInsertContextEntry

```
int          tfInsertContextEntry
(
ttSnmPVacmContextTable  *x
);
```

#### Function Description

This function inserts the new entry into the linked list of ttSnmPVacmContextTable's. The struct ttSnmPVacmAccessTable is defined in trsnmpv3.h:

```
typedef struct tsSnmPVacmContextTable
{
    struct tsSnmPVacmContextTable *svctNext;
    int svctNameLen;
    char svctName[32];
} ttSnmPVacmContextTable;
```

The table index consists of svctName. Note that the user should specify index field (together with svctNameLen). It is a responsibility of a calling function to allocate memory for x. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
x	Input parameter, pointer to ttSnmPVacmContextTable, with svctName and svctNameLen filled in.

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.
TM_WRONG_CONTEXTNAME_LENGTH	Failure, svctNameLen is not between 0 and 32.
TM_CONTEXTNAME_NOT_PRINTABLE	Failure, svctName contains unprintable characters.
TM_ALLOCATION_ERROR	Failure to allocate memory for snmpVacmContextEntryStart

### 4.3.26 `tfInsertSec2GroupEntry`

```
int                tfInsertSec2GroupEntry
(
ttSnmPVacmSec2GroupTable    *x
);
```

#### Function Description

This function either inserts the new entry into the linked list of `ttSnmPVacmSec2GroupTable`'s, if the compound table index does not coincide with any index in the entries in the linked list, or replaces the old entry with the new one, if the identical index is found. The struct `ttSnmPVacmAccessTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPVacmSec2GroupTable
{
    struct tsSnmPVacmSec2GroupTable *svgtNext;
    int svgtModel;
    int svgtNameLen;
    int svgtGroupNameLen;
    int svgtStorageType;
    int svgtStatus;
    char svgtName[32];
    char svgtGroupName[32];
} ttSnmPVacmSec2GroupTable;
```

The table index consists of `svgtModel`, and `svgtName` (in that order). Note that the user should specify both index fields (together with `svgtNameLen`), and `svgtGroupName`, `svgtGroupNameLen` and `svgtStatus` values since they do not have `DEFVAL` clauses. If the user does not specify other fields, they will be assigned their default values, per RFC 2575. It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

##### Parameter

`x`

##### Description

Input parameter, pointer to `ttSnmPVacmSec2GroupTable`, with the fields `svgtModel`, `svgtName`, `svgtNameLen`, `svgtGroupName`, `svgtGroupNameLen` and `svgtStatus` filled in.

#### Returns

##### Value

`TM_SNMPI_NOERROR`

`TM_ALLOCATION_ERROR`

`TM_WRONG_SECURITYMODEL`

`TM_WRONG_SECURITYNAME_LENGTH`

`TM_SECURITYNAME_NOT_PRINTABLE`

`TM_WRONG_GROUPNAME_LENGTH`

`TM_GROUPNAME_NOT_PRINTABLE`

`TM_WRONG_STATUS`

`TM_ALLOCATION_ERROR`

##### Meaning

Success.

Failure to allocate memory for the agent global variables.

Failure, `svgtModel < 1`.

Failure, `svgtNameLen` is not between 1 and 32.

Failure, `svgtName` contains unprintable characters.

Failure, `svgtGroupNameLen` is not between 1 and 32.

Failure, `svgtGroupName` contains unprintable characters.

Failure, `svgtStatus` is not 1, or 2, or 3.

Failure to allocate memory for `snmpVacmSec2GroupEntryStart`

### 4.3.27 `tfInsertUsmEntry`

```
int          tfInsertUsmEntry
(
ttSnmPUsmUserTable  *x
);
```

#### Function Description

This function either inserts the new entry into the linked list of `ttSnmPUsmUserTable`'s, if the compound table index does not coincide with any index in the entries in the linked list, or replaces the old entry with the new one, if the identical index is found. The struct `ttSnmPUsmUserTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPUsmUserTable
{
    struct tsSnmPUsmUserTable *suutNext;
    oid suutCloneFrom[TM_SNMP_MAX_NAME_LEN];
    oid suutAuthProtocol[TM_SNMP_MAX_NAME_LEN];
    oid suutPrivProtocol[TM_SNMP_MAX_NAME_LEN];
    int suutSecurityNameLen;
    int suutCloneFromLen;
    int suutAuthProtocolLen;
    int suutAuthKeyChangeLen;
    int suutOwnAuthKeyChangeLen;
    int suutPrivProtocolLen;
    int suutPrivKeyChangeLen;
    int suutOwnPrivKeyChangeLen;
    int suutPublicLen;
    int suutStorageType;
    int suutStatus;
    char suutSecurityName[32];
    char suutAuthKeyChange[32];
    char suutOwnAuthKeyChange[32];
    char suutPrivKeyChange[32];
    char suutOwnPrivKeyChange[32];
    char suutUuPublic[32];
    char suutName[32];
    unsigned char suutEngineId[20];
    unsigned char suutEngineIdLen;
    unsigned char suutNameLen;
    char suutPad[2];
} ttSnmPUsmUserTable;
```

The table index consists of `suutEngineId` and `suutName` (in that order). Note that the user should specify both index fields (together with their length values), and `suutSecurityName`, `suutSecurityNameLen`, `suutCloneFrom`, `suutCloneFromLen` and `suutStatus` values since they do not have DEFVAL clauses. If the user does not specify other fields, they will be assigned their default values, per RFC 2574. It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

**Parameter**  
`x`

#### Description

Input parameter, pointer to `ttSnmPUsmUserTable`, with the fields `suutEngineID`, `suutEngineIdLen`, `suutName`, `suutNameLen`, `suutSecurityName`, `suutSecurityNameLen`, `suutCloneFrom`, `suutCloneFromLen` and `suutStatus` filled in.

**Returns**

<b>Value</b>	<b>Meaning</b>
TM_SNMPAPI_NOERROR	Success.
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.
TM_ZERO_ENGINEID_LENGTH	Failure, suutEngineIdLen is 0.
TM_WRONG_USERNAME_LENGTH	Failure, suutNameLen is not between 1 and 32.
TM_USERNAME_NOT_PRINTABLE	Failure, suutName contains unprintable characters.
TM_ZERO_SECURITYNAME_LENGTH	Failure, suutSecurityNameLen is 0.
TM_SECURITYNAME_NOT_PRINTABLE	Failure, suutSecurityName contains unprintable characters.
TM_ZERO_CLONEFROM_LENGTH	Failure, suutCloneFromLen is 0.
TM_WRONG_PUBLIC_LENGTH	Failure, suutPublicLen is not between 0 and 32.
TM_PUBLIC_NOT_PRINTABLE	Failure, suutPublic contains unprintable characters.
TM_UNKNOWN_STORAGE_TYPE	Failure, suutStorageType is not 0 or between 1 and 5.
TM_WRONG_STATUS	Failure, suutStatus is not 1, or 2, or 3.
TM_ALLOCATION_ERROR	Failure to allocate memory for snmpUsmUserEntryStart

### 4.3.28 tfInsertVtfEntry

```
int          tfInsertVtfEntry
(
ttSnmPVacmViewTreeFamilyTable *x
);
```

#### Function Description

This function either inserts the new entry into the linked list of `ttSnmPVacmViewTreeFamilyTable`'s, if the compound table index does not coincide with any index in the entries in the linked list, or replaces the old entry with the new one, if the identical index is found. The struct `ttSnmPVacmViewTreeFamilyTable` is defined in `trsnmpv3.h`:

```
typedef struct tsSnmPVacmViewTreeFamilyTable
{
    struct tsSnmPVacmViewTreeFamilyTable *svvtNext;
    oid svvtSubtree[TM_SNMP_MAX_NAME_LEN];
    int svvtNameLen;
    int svvtSubtreeLen;
    int svvtMaskLen;
    int svvtType;
    int svvtStorageType;
    int svvtStatus;
    char svvtName[32];
    unsigned char svvtMask[16];
} ttSnmPVacmViewTreeFamilyTable;
```

The table index consists of `svvtName` and `svvtSubtree` (in that order). Note that the user should specify both index fields (together with their length values) and `svvtStatus` value since they do not have `DEFVAL` clauses. If the user does not specify other fields, they will be assigned their default values, per RFC 2575. It is a responsibility of a calling function to allocate memory for `x`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<code>x</code>	Input parameter, pointer to <code>ttSnmPVacmViewTreeFamilyTable</code> , with the fields <code>svvtName</code> , <code>svvtNameLen</code> , <code>svvtSubtree</code> , <code>svvtSubtreeLen</code> and <code>svvtStatus</code> filled in.

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_WRONG_VTFNAME_LENGTH</code>	Failure, <code>svvtNameLen</code> is not between 0 and 32.
<code>TM_VTFNAME_NOT_PRINTABLE</code>	Failure, <code>svvtName</code> contains unprintable characters.
<code>TM_WRONG_TYPE</code>	Failure, <code>svvtType</code> is not 0, or 1, or 2.
<code>TM_WRONG_STATUS</code>	Failure, <code>svvtStatus</code> is not 1, or 2, or 3
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for <code>snmpVacmViewTreeFamilyEntryStart</code>

### 4.3.29 tfModifyCommunity

```
int          tfModifyCommunity
(
int          which,
char        *target,
char        *source
);
```

#### Function Description

This function lets the user modify a community string in the list of communities. Two pools of communities, the sizes of `TM_SNMP_MAX_NUM_READ_COMMUNITIES` and `TM_SNMP_MAX_NUM_WRITE_COMMUNITIES`, are defined. Maximum length of the community string in each pool is set as `TM_SNMP_MAX_COMMUNITY_LEN`. The user must indicate the pool by setting `which` parameter. The user must also specify the community to be modified, `target`, and the new community value, `source`. It is a responsibility of a calling function to allocate memory for `target` and `source`. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

##### Parameter

*which*

*target*

*source*

##### Description

Input parameter, an integer value taking on `TM_READ_COMMUNITY_FLAG`, or `TM_WRITE_COMMUNITY_FLAG`

Input parameter, community string to be modified.

Input parameter, a new community string.

#### Returns

##### Value

`TM_SNMPAPI_NOERROR`

`TM_ALLOCATION_ERROR`

`TM_RECORD_LOCKED`

`TM_RECORD_NOTFOUND`

`TM_UNKNOWN_COMMUNITY_FLAG`

`TM_COMMUNITY_TOOLONG`

`TM_COMMUNITY_NOT_PRINTABLE`

##### Meaning

Success.

Failure to allocate memory for the agent global variables.

Failure, communities list is locked.

Failure, target community is not found

Failure, which value is not defined

Failure, source is too long

Failure, source contains unprintable characters

### 4.3.30 tfModifySysContact

```
int tfModifySysContact
(
char *sysContact
);
```

#### Function Description

This function lets the user modify MIB-II sysContact object instance.

By definition, sysContact is a string containing the textual identification of the contact person for the managed node, together with information on how to contact this person. Its length must not exceed 255 bytes. The agent initializes it to the default value of the octet string tvSnmppSysContact, defined in trecfg.h. This function allows the user to change this default value. It is the responsibility of the calling function to allocate memory for sysContact. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>sysContact</i>	Input parameter, an octet string value of sysContact MIB-II object

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.
TM_SYSCONTACT_TOOLONG	Failure, sysContact contains too many characters.
TM_SYSCONTACT_NOT_PRINTABLE	Failure, sysContact contains unprintable characters.

### 4.3.31 tfModifySysLocation

```
int tfModifySysLocation
(
char *sysLocation
);
```

#### Function Description

This function lets the user modify MIB-II sysLocation object instance.

By definition, sysLocation is a string containing the physical location of the system running the SNMP agent. Its length must not exceed 255 bytes. The agent initializes it to the default value of the octet string tvSnmpSysLocation, defined in trcfg.h. This function allows the user to change this default value. It is the responsibility of the calling function to allocate memory for sysLocation. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
-----------	-------------

<i>sysLocation</i>	Input parameter, an octet string value of sysLocation MIB-II object
--------------------	---

#### Returns

Value	Meaning
-------	---------

TM_SNMPAPI_NOERROR	Success
--------------------	---------

TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.
---------------------	--

TM_SYSLOCATION_TOOLONG	Failure, sysLocation contains too many characters.
------------------------	--

TM_SYSLOCATION_NOT_PRINTABLE	Failure, sysLocation contains unprintable characters.
------------------------------	---

### 4.3.32 tfModifySysName

```
int tfModifySysName
(
char *sysName
);
```

#### Function Description

This function lets the user modify MIB-II sysName object instance.

By definition, sysName is a string containing an administratively-assigned name for the system running the SNMP agent. By convention, this should be its fully-qualified domain name. Its length must not exceed 255 bytes. The agent initializes it to the default value of the octet string tvSnmppSysName, defined in trcfg.h. This function allows the user to change this default value. It is the responsibility of the calling function to allocate memory for sysName. The user can call this function any time prior to starting the agent, or during the runtime.

#### Parameters

Parameter	Description
<i>sysName</i>	Input parameter, an octet string value of sysName MIB-II object

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables.
TM_SYSNAME_TOOLONG	Failure, sysName contains too many characters.
TM_SYSNAME_NOT_PRINTABLE	Failure, sysName contains unprintable characters.

#### **4.3.33 tfModifyTrapEntry**

---

*tfModifyTrapEntry has been deprecated API.  
Please use tfNgModifyTrapEntry instead.*

---

### 4.3.34 tNgAddTrapEntry

```
int tNgAddTrapEntry
(
ttNgSnmptTrapParms *entry
);
```

#### Function Description

This function lets the user add a new entry to the list of trap sinks. The pool of trap sinks of size TM\_SNMP\_MAX\_NUM\_TRAP\_SINKS is defined. By default, when the agent starts, if the pool is empty, the agent initializes with one default entry. It is a responsibility of a calling function to allocate memory for entry. The user can call this function any time prior to starting the agent, or during the runtime. The input argument is the pointer to the struct ttNgSnmptTrapParms defined in trconsts.h:

```
typedef struct tsNgSnmptTrapParms
{
    struct sockaddr_storage stpDestIp; /* trap manager's IP address */
    struct sockaddr_storage stpSrcIp; /* Agent IP address */

    /* enterprise number assigned to the organization by IETF */
    unsigned long stpEnterpriseNum;
    unsigned long stpEnterpriseLen; /* length of stpEnterprise OID */

    /* trap enterprise field of SNMPv1 trap. It can be of any length between 8 and
    * TM_SNMP_MAX_NAME_LEN. The sub-oids 0 to 6 are pre-defined as
    * 1.3.6.1.4.1.stpEnterpriseNum
    */
    oid stpEnterprise[TM_SNMP_MAX_NAME_LEN];

    int stpCommunityLen; /* community string length */
    int stpSocket; /* agent's trap socket (assigned by system) */

    /* the community string sent with SNMPv1 and SNMPv2 traps */
    unsigned char stpCommunity[((TM_SNMP_MAX_NAME_LEN + 3) / 4) * 4];
} ttNgSnmptTrapParms;
```

Since there are no defaults, the user must specify all members' values (except stpSocket, and possibly stpEnterprise). If any of the values is not set, the trap may be either not be received by the trap manager, or, if received, not properly decoded.

#### Parameters

Parameter	Description
<i>Entry</i>	Pointer to ttNgSnmptTrapParms with the members filled in on input.

#### Returns

Value	Meaning
TM_SNMPAPI_NOERROR	Success.
TM_ALLOCATION_ERROR	Failure to allocate memory for the agent global variables
TM_RECORD_LOCKED	Failure, trap sinks list is locked
TM_NOMORE_TRAPENTRIES	Failure, trap sinks list is full
TM_PORT_TOOLARGE	Failure, port specified in stpDestIp is larger than 65535
TM_COMMUNITY_TOOLONG	Failure, stpCommunity is too long
TM_COMMUNITY_NOT_PRINTABLE	Failure, stpCommunity contains unprintable characters
TM_EMFILE	Failure, no more sockets are available
TM_ENOBUFS	Failure, insufficient user memory to complete the operation

**Example Code**

The following example code shows how you could call `tfNgAddTrapEntry` to override the default trap destination IP address prior to starting the SNMP Agent. This example assumes that you are running the SNMP Agent in blocking mode as a separate task:

```
void snmpdTask(void *data)
{
    int ret;
    struct sockaddr_in *addrPtr;
    ttNgSnmpTrapParms myTrap;

    tm_snmp_strcpy(myTrap.stpCommunity, "trap comm");
    myTrap.stpCommunityLen = strlen(myTrap.stpCommunity);

    addrPtr = (struct sockaddr_in *) &myTrap.stpDestIp;
    addrPtr->sin_family = AF_INET;
    addrPtr->sin_len = sizeof(struct sockaddr_in);

    addrPtr->sin_port = htons(162);
    retVar = inet_pton(addrPtr->sin_family,
        "12.99.240.220",
        &addrPtr->sin_addr);

    addrPtr = (struct sockaddr_in *) &MyPtr->stpSrcIp;
    addrPtr->sin_addr.s_addr = tfGetRouterId();

    trapParmsPtr->stpEnterpriseNum = TM_DEFAULT_ENTERPRISE_NUMBER;
    trapParmsPtr->stpEnterpriseLen = TM_DEFAULT_ENTERPRISE_LENGTH;

    retVar = tfNgAddTrapEntry(&myTrap);

    ret = tfSnmpdMain(TM_BLOCKING_ON, 0);
}
```

### 4.3.35 tNngDeleteTrapEntry

```
int tNngDeleteTrapEntry
(
ttNgSnmptTrapParms *entry
);
```

#### Function Description

This function lets the user delete an entry from the list of trap sinks. The pool of trap sinks of size `TM_SNMP_MAX_NUM_TRAP_SINKS` is defined. By default, when the agent starts, if the pool is empty, the agent initializes with one default entry. It is a responsibility of a calling function to allocate memory for entry. The user can call this function any time prior to starting the agent, or during the runtime. If this entry is the only one left in the pool, it can not be deleted. The input argument is the pointer to the struct `ttNgSnmptTrapParms` defined in `trconsts.h`:

```
typedef struct tsNgSnmptTrapParms
{
    struct sockaddr_storage stpDestIp; /* trap manager's IP address */
    struct sockaddr_storage stpSrcIp; /* Agent IP address */

    /* enterprise number assigned to the organization by IETF */
    unsigned long stpEnterpriseNum;
    unsigned long stpEnterpriseLen; /* length of stpEnterprise OID */

    /* trap enterprise field of SNMPv1 trap. It can be of any length between 8 and
    * TM_SNMP_MAX_NAME_LEN. The sub-oids 0 to 6 are pre-defined as
    * 1.3.6.1.4.1.stpEnterpriseNum
    */
    oid stpEnterprise[TM_SNMP_MAX_NAME_LEN];

    int stpCommunityLen; /* community string length */
    int stpSocket; /* agent's trap socket (assigned by system) */

    /* the community string sent with SNMPv1 and SNMPv2 traps */
    unsigned char stpCommunity[((TM_SNMP_MAX_NAME_LEN + 3) / 4) * 4];
} ttNgSnmptTrapParms;
```

Note, that the user must specify the following members of `ttNgSnmptTrapParms`: `stpCommunityLen`, `stpCommunity`, and `stpDestIp` inside entry. The reason is, there might be multiple entries in the pool containing any subset of these four parameters, e.g. different entries in the pool specify sending traps to the same address with the same communities, but to different ports.

#### Parameters

Parameter	Description
<i>entry</i>	Pointer to <code>ttNgSnmptTrapParms</code> with the members <code>stpCommunityLen</code> , <code>stpCommunity</code> , <code>stpDestIp</code> filled in on input.

#### Description

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_RECORD_LOCKED</code>	Failure, trap sinks list is locked.
<code>TM_RECORD_NOTFOUND</code>	Failure, entry is not found.
<code>TM_DEFAULT_NOTDELETED</code>	Failure, the last trap entry cannot be deleted

### 4.3.36 ttNgDisplayTrapEntry

```
int ttNgDisplayTrapEntry
(
ttNgSnmpTrapParms *entryPtr,
unsigned int i
);
```

#### Function Description

This function lets the user display an entry from the list of trap sinks. It might be convenient to inspect an existing entry prior to deleting one, or adding one, or for any other purpose. The pool of trap sinks of size `TM_SNMP_MAX_NUM_TRAP_SINKS` is defined. The user must specify the entry number, `i` (0-based). It is a responsibility of the calling function to allocate memory to hold the `ttNgSnmpTrapParms` pointed to by `entryPtr` prior to calling this function. The user can call this function any time prior to starting the agent, or during the runtime. The type `ttNgSnmpTrapParms` is defined in `trconsts.h`:

```
typedef struct tsNgSnmpTrapParms
{
    struct sockaddr_storage stpDestIp; /* trap manager's IP address */
    struct sockaddr_storage stpSrcIp; /* Agent IP address */

    /* enterprise number assigned to the organization by IETF */
    unsigned long stpEnterpriseNum;
    unsigned long stpEnterpriseLen; /* length of stpEnterprise OID */

    /* trap enterprise field of SNMPv1 trap. It can be of any length between 8
    * and TM_SNMP_MAX_NAME_LEN. The sub-oids 0 to 6 are pre-defined as
    * 1.3.6.1.4.1.stpEnterpriseNum
    */
    oid stpEnterprise[TM_SNMP_MAX_NAME_LEN];

    int stpCommunityLen; /* community string length */
    int stpSocket; /* agent's trap socket (assigned by system) */

    /* the community string sent with SNMPv1 and SNMPv2 traps */
    unsigned char stpCommunity[((TM_SNMP_MAX_NAME_LEN + 3) / 4) * 4];
} ttNgSnmpTrapParms;
```

#### Parameters

Parameter	Meaning
<i>entryPtr</i>	Pointer to memory allocated by the user, which is populated by this function with information copied from the trap entry at the specified index.
<i>i</i>	Index of the trap entry to be displayed

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success.
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_RECORD_NOTFOUND</code>	Failure, trap entry was not found.

### 4.3.37 `tfNgModifyTrapEntry`

```
int tfNgModifyTrapEntry
(
ttNgSnmptTrapParms *target,
ttNgSnmptTrapParms *source
);
```

#### Function Description

This is the list of trap sinks. The pool of trap sinks of size `TM_SNMP_MAX_NUM_TRAP_SINKS` is defined. It is a responsibility of a calling function to allocate memory for `target` and `source`. The user can call this function any time prior to starting the agent, or during the runtime. The input argument is the pointer to the struct `ttNgSnmptTrapParms` defined in `trconsts.h`:

```
typedef struct tsNgSnmptTrapParms
{
    struct sockaddr_storage stpDestIp; /* trap manager's IP address */
    struct sockaddr_storage stpSrcIp; /* Agent IP address */

    /* enterprise number assigned to the organization by IETF */
    unsigned long stpEnterpriseNum;
    unsigned long stpEnterpriseLen; /* length of stpEnterprise OID */

    /* trap enterprise field of SNMPv1 trap. It can be of any length between 8
    * and TM_SNMP_MAX_NAME_LEN. The sub-oids 0 to 6 are pre-defined as
    * 1.3.6.1.4.1.stpEnterpriseNum
    */
    oid stpEnterprise[TM_SNMP_MAX_NAME_LEN];

    int stpCommunityLen; /* community string length */
    int stpSocket; /* agent's trap socket (assigned by system) */

    /* the community string sent with SNMPv1 and SNMPv2 traps */
    unsigned char stpCommunity[((TM_SNMP_MAX_NAME_LEN + 3) / 4) * 4];
} ttNgSnmptTrapParms;
```

Note, that the user should specify the following members of `ttNgSnmptTrapParms`: `stpCommunityLen`, `stpCommunity` and `stpDestIp` inside `target`. The reason is, there might be multiple entries in the pool containing any subset of these four parameters, e.g. different entries in the pool specify sending traps to the same address with the same communities, but to different ports. The user must also specify the same members inside `source`, since there are no defaults.

#### Parameters

Parameter	Description
<i>target</i>	Pointer to <code>ttNgSnmptTrapParms</code> to be modified, with the members <code>stpCommunityLen</code> , <code>stpCommunity</code> and <code>stpDestIp</code> filled in on input.
<i>source</i>	Pointer to the new <code>ttNgSnmptTrapParms</code> with the members <code>stpCommunityLen</code> , <code>stpCommunity</code> and <code>stpDestIp</code> filled in on input.

#### Returns

Value	Meaning
<code>TM_SNMPAPI_NOERROR</code>	Success
<code>TM_ALLOCATION_ERROR</code>	Failure to allocate memory for the agent global variables.
<code>TM_RECORD_LOCKED</code>	Failure, trap sinks list is locked.
<code>TM_RECORD_NOTFOUND</code>	Failure, target is not found.
<code>TM_PORT_TOOLARGE</code>	Failure, trap port is larger than 65535
<code>TM_COMMUNITY_TOOLONG</code>	Failure, source community is too long
<code>TM_COMMUNITY_NOT_PRINTABLE</code>	Failure, source community contains unprintable characters

## 4.4 Protocol stack-specific functions

This section is just included for completeness, since these APIs are the interface between the SNMP Agent and the Treck stack. Normally, you will not need to modify these functions, but if you do, they are all implemented in the file `trlocal.c`.

### 4.4.1 `tfGetNumberIfs`

```
int                tfGetNumberIfs
(
);
```

#### Function Description

This function retrieves a number of interfaces in the system.

#### Parameters

None

#### Returns

Number of interfaces in the system

#### 4.4.2 tflcmpGroupInfoGet

```
long
*tflcmpGroupInfoGet
(
unsigned char                what
);
```

##### Function Description

This function retrieves a value of an object in icmp group. The calling function must specify which object by using the “what” argument.

##### Parameters

###### Parameter

*what*

###### Description

Magic number identifying a member of MIB-II icmp group, one of the following defined in trconsts.h:

###### *what Parameter Descriptions*

TM_MIB_ICMPINMSGS:	The total number of ICMP messages which the entity received, including all those counted by TM_MIB_ICMPINERRORS below.
TM_MIB_ICMPINERRORS:	The number of ICMP messages which the entity received but but determined as having ICMP-specific errors (bad ICMP checksum, bad length, etc.).
TM_MIB_ICMPINDESTUNREACHS:	The number of ICMP Destination Unreachable messages received.
TM_MIB_ICMPINTIMEEXCDS:	The number of ICMP Time Exceeded messages received.
TM_MIB_ICMPINPARMPROBS:	The number of ICMP Parameter Problem messages received.
TM_MIB_ICMPINSRCQUENCHS:	The number of ICMP Source Quench messages received,
TM_MIB_ICMPINREDIRECTS:	The number of ICMP Redirect messages received.
TM_MIB_ICMPINECHOS:	The number of ICMP Echo (request) messages received.
TM_MIB_ICMPINECHOREPS:	The number of ICMP Echo Reply messages received.
TM_MIB_ICMPINTIMESTAMPS:	The number of ICMP Timestamp (request) messages received.
TM_MIB_ICMPINTIMESTAMPREPS:	The number of ICMP Timestamp Reply messages received.
TM_MIB_ICMPINADDRMASKS:	The number of ICMP Address Mask Request messages received.
TM_MIB_ICMPINADDRMASKREPS:	The number of ICMP Address Mask Reply messages received.
TM_MIB_ICMPOUTMSGS:	The total number of ICMP messages which this entity attempter to send. Note that this counter include all those counted by TM_MIB_ICMPOUTERRORS below.
TM_MIB_ICMPOUTERRORS:	The number of ICMP messages which this entity did not send due to the problems discovered within ICMP such as lack of buffers. This value should not include errors discovered outside the ICMP layer such as the inability of IP to route the resultant datagram. In some implementations there may be no types of error which contribute to this counter's value.
TM_MIB_ICMPOUTDESTUNREACHS:	The number of ICMP Destination Unreachable messages sent.
TM_MIB_ICMPOUTTIMEEXCDS:	The number of ICMP Time Exceeded messages sent.
TM_MIB_ICMPOUTPARMPROBS:	The number of ICMP Parameter Problem messages sent.
TM_MIB_ICMPOUTSRCQUENCHS:	The number of ICMP Source Quench messages sent.
TM_MIB_ICMPOUTREDIRECTS:	The number of ICMP Redirect messages sent. For a host, this object will always be zero since hosts do not send redirects.
TM_MIB_ICMPOUTECHOS:	The number of ICMP Echo (request) messages sent.
TM_MIB_ICMPOUTECHOREPS:	The number of ICMP Echo Reply messages sent.

TM\_MIB\_ICMPOUTTIMESTAMPS: The number of ICMP Timestamp (request) messages sent.  
TM\_MIB\_ICMPOUTTIMESTAMPREPS: The number of ICMP Timestamp Reply messages sent.  
TM\_MIB\_ICMPOUTADDRMASKS: The number of ICMP Address Mask Request messages sent.  
TM\_MIB\_ICMPOUTADDRMASKREPS: The number of ICMP Address Mask Reply messages sent.

**Returns**

**Value**

NULL

!=NULL

**Meaning**

The object value was not found.

Pointer to the found object value.

### 4.4.3 tflfGroupInfoGet

```

long          *tflfGroupInfoGet
(
int           interfaceIndex,
unsigned char what,
int          *lengthPtr
);

```

#### Function Description

This function retrieves parameters and statistics values on a given interface. The calling function must specify `interfaceIndex` and `what` parameters. The length of a parameter/statistics value is ignored on input and returned in `*lengthPtr`.

#### Parameters

Parameter	Description
<i>interfaceIndex</i>	Interface number (1-based index).
<i>lengthPtr</i>	Pointer to the parameter/statistics value length on return.
<i>what</i>	Magic number identifying a member of MIB-II interface group, one of the following defined in <code>trconsts.h</code> :

#### *what* parameter description

TM_MIB_IFINDEX:	The interface number (1-based index).
TM_MIB_IFDESCR:	Information about the interface, including name of manufacturer, product name, and version of the hardware interface.
TM_MIB_IFTYPE:	Type of interface, distinguished according to the physical/link protocol(s).
TM_MIB_IFMTU:	The size of the largest protocol data unit, in bytes, that can be sent/received on the interface.
TM_MIB_IFSPEED:	An estimate of the interface's current data rate capacity.
TM_MIB_IFPHYSADDRESS:	The interface's address at the protocol layer immediately below the network layer (i.e. Ethernet address, if you are using the Ethernet link layer).
TM_MIB_IFADMINSTATUS:	Desired interface state: up(1), down(2), testing(3).
TM_MIB_IFOPERSTATUS:	Current operational interface state: up(1), down(2), testing(3).
TM_MIB_IFLASTCHANGE:	Value of <code>sysUpTime</code> at the time the interface entered its current operational state.
TM_MIB_IFINOCETS:	Total number of bytes received on the interface, including framing characters.
TM_MIB_IFINUCASTPKTS:	Number of subnetwork-unicast packets delivered to a higher-layer protocol.
TM_MIB_IFINNUCASTPKTS:	Number of non-unicast packets (i.e. broadcast, multicast) delivered to a higher-layer protocol.
TM_MIB_IFINDISCARDS:	Number of inbound packets discarded, even though no errors had been detected, to prevent their being delivered to a higher-layer protocol (i.e. buffer overflow).
TM_MIB_IFINERRORS:	Number of inbound packets that contained errors preventing them from being delivered to a higher-layer protocol.
TM_MIB_IFINUNKNOWNPROTOS:	Number of inbound packets that were discarded because of an unknown or unsupported protocol.
TM_MIB_IFOUTOETS:	Total number of bytes transmitted on the interface, including framing characters.
TM_MIB_IFOUTUCASTPKTS:	Total number of packets that higher-level protocols requested be transmitted to a subnetwork-unicast address, including those that were discarded or otherwise not sent.

TM_MIB_IFOUTNUCASTPKTS:	Total number of packets that higher-level protocols requested be transmitted to a non-unicast address (i.e. broadcast, multicast), including those that were discarded or otherwise not sent.
TM_MIB_IFOUTDISCARDS:	Number of outbound packets discarded even though no errors had been detected to prevent their being transmitted (i.e. buffer overflow).
TM_MIB_IFOUTERRORS:	Number of outbound packets that could not be transmitted because of errors.
TM_MIB_IFOUTQLEN:	Length of the output packet queue.
TM_MIB_IFSPECIFIC:	OID reference to MIB definitions specific to the particular media being used to realize the interface.

**Returns**

**Value**

NULL

!=NULL

**Meaning**

The parameter/statistics value was not found.

Pointer to the found parameter/statistics value.

#### 4.4.4 `tfIpAddrTableEntryGet`

```
long          *tfIpAddrTableEntryGet
(
int          exact,
ttMib2IpAddrEntry *ip_addr_table
);
```

##### Function Description

This function retrieves the values in a particular row of `ipAddrTable`. The `ttMib2IpAddrEntry` is defined in `trmib2.h` as follows:

```
typedef struct tsMib2IpAddrEntry
{
    unsigned long ipAdEntAddr;
    unsigned long ipAdEntNetMask;
    unsigned int ipAdEntIfIndex;
    unsigned int ipAdEntBcastAddr;
    unsigned int ipAdEntReasmMaxSize;
} ttMib2IpAddrEntry;
```

The rows in `ipAddrTable` are indexed by `ipAdEntAddr`. On input the calling function must specify this field to indicate the row for which the values of `ipAdEntIfIndex`, `ipAdEntNetMask`, `ipAdEntBcastAddr` and `ipAdEntReasmMaxSize` to be returned. The calling function must also specify if an exact match is requested by setting the value of `exact` to `TRUE` if so, and to `FALSE` otherwise. This function performs the search of `ipAddrTable` as follows:

- 1) if `exact` is `TRUE`, and the row with the corresponding `ipAdEntAddr` is found, fill in the remaining fields and return;
- 2) if `exact` is `TRUE`, and the row with the corresponding `ipAdEntAddr` is not found, return;
- 3) if `exact` is `FALSE`, and the row lexicographically next to the requested one is found, fill in all fields, including `ipAdEntAddr`, and return;
- 4) if `exact` is `FALSE`, and the row lexicographically next to the requested one is not found, return.

It is the responsibility of this function to sort the rows in ascending order of `ipAdEntAddr`, if not already sorted. It is the responsibility of a calling function to allocate memory for `ip_addr_table`.

##### Parameters

Parameter	Description
<i>exact</i>	The flag indicating whether an exact match is requested ( <code>TRUE</code> or <code>FALSE</code> )
<i>ip_addr_table</i>	Pointer to <code>ttMib2IpAddrEntry</code> with the <code>ipAdEntAddr</code> field filled in on input; pointer to <code>ttMib2IpAddrEntry</code> with all members filled on output

##### Returns

Value	Meaning
0	<i>exact</i> = <code>TRUE</code> and the row was not found, OR <i>exact</i> = <code>FALSE</code> and the row lexicographically following the requested one was not found.
1	<i>exact</i> = <code>TRUE</code> and the row was found, OR <i>exact</i> = <code>FALSE</code> and the row lexicographically following the requested one was found.

**Example**

Suppose ipAddrTable contains the following entries (only address is shown):

**address**

10.1.23.5

128.97.88.44

192.3.11.117

201.2.11.117

If this function is called with `exact = TRUE` and the `ipAdEntAddr` field 128.97.88.44, the values from the second row are filled in, and 1 is returned.

If this function is called with `exact = TRUE` and the `ipAdEntAddr` field 128.97.88.34, 0 is returned.

If this function is called with `exact = FALSE` and the `ipAdEntAddr` field 0.0.0.0, all values from the first row are filled in, and 1 is returned.

If this function is called with `exact = FALSE` and the `ipAdEntAddr` field 193.0.0.0, all values from the last row are filled in, and 1 is returned.

If this function is called with `exact = FALSE` and the `ipAdEntAddr` field 203.0.0.0, 0 is returned.

#### 4.4.5 tflpGroupInfoGet

```
long                *tflpGroupInfoGet
(
unsigned char      what
);
```

#### Function Description

This function retrieves a value of an object in ip group. The calling function must specify which object by using the “what” argument.

#### Parameters

##### Parameter

*what*

##### Description

Magic number identifying a member of MIB-II ip group, one of the following defined in trconst.h:

##### *what* parameter description

TM\_MIB\_IPFORWARDING:

The indication of whether this entity is acting as an IP gateway in respect to the forwarding of datagrams received by, but not addressed to, this entity. IP gateways forward datagrams. IP hosts do not (except those source-routed via the host) Note that for some managed nodes, this object may take on only a subset of the values possible. Accordingly, it is appropriate for an agent to return a ‘badValue’ response if a management station attempts to change this object to an inappropriate value.

TM\_MIB\_IPDEFAULT\_TTL:

The default value inserted into the Time-To-Live field of the IP header of datagrams originated at this entity, whenever a TTL value is not supplied by the transport layer protocol.

TM\_MIB\_IPINRECEIVES:

The total number of input datagrams received from interfaces, including those received in error.

TM\_MIB\_IPINHDRERRORSS:

The number of input datagrams discarded due to errors in their IP headers, including bad checksums, version number mismatch, other format errors, time-to-live exceeded, errors discovered in processing their IP options, etc.

TM\_MIB\_IPINADDRERRORS:

The number of input datagrams discarded because the IP address in their IP header’s destination field was not a valid address to be received at this entity. This count includes invalid addresses (e.g., 0.0.0.0) and addresses of unsupported Classes (e.g., Class E). For entities which are not IP Gateways and therefore do not forward datagrams, this counter includes datagrams discarded because the destination address was not a local address.

TM\_MIB\_IPFORWDATAGRAMS:

The number of input datagrams for which this entity was not their final IP destination, as a result of which an attempt was made to find a route to forward them to that final destination. In entities which do not act as IP Gateways, this counter will include only those packets which were Source-Routed via this entity, and the Source-Route option processing was successful.

TM\_MIB\_IPINUNKNOWNPROTOS:

The number of locally-addressed datagrams received successfully but discarded because of an unknown or unsupported protocol.

TM\_MIB\_IPINDISCARDS:

The number of input IP datagrams for which no problems were encountered to prevent their continued processing, but which were discarded (e.g., for lack of buffer space). Note that this counter does not include any datagrams discarded while awaiting re-assembly.

TM\_MIB\_IPINDELIVERS:

The total number of input datagrams successfully delivered to IP user-protocols (including ICMP).

TM_MIB_IPOUTREQUESTS:	The total number of IP datagrams which local IP user-protocols (including ICMP) supplied to IP in requests for transmission. Note that this counter does not include any datagrams counted in ipForwDatagrams.
TM_MIB_IPOUTDISCARDS:	The number of output IP datagrams for which no problem was encountered to prevent their transmission to their destination, but which were discarded (e.g., for lack of buffer space). Note that this counter would include datagrams counted in ipForwDatagrams if any such packets met this (discretionary) discard criterion.
TM_MIB_IPOUTNOROUTES:	The number of IP datagrams discarded because no route could be found to transmit them to their destination. Note that this counter includes any packets counted in ipForwDatagrams which meet this 'no-route' criterion. Note that this includes any datagrams which a host cannot route because all of its default gateways are down.
TM_MIB_IPREASMTIMEOUT:	The maximum number of seconds which received fragments are held while they are awaiting reassembly at this entity.
TM_MIB_IPREASMREQDS:	The number of IP fragments received which needed to be reassembled at this entity.
TM_MIB_IPREASMOKS:	The number of IP datagrams successfully reassembled.
TM_MIB_IREASMFAILS:	The number of failures detected by the IP reassembly algorithm (for whatever reason: timed out, errors, etc). Note that this is not necessarily a count of discarded IP fragments since some algorithms (notably the algorithm in RFC 815) can lose track of the number of fragments by combining them as they are received.
TM_MIB_IPFRAGOKS:	The number of IP datagrams that have been successfully fragmented at this entity.
TM_MIB_IPFRAGFAILS:	The number of IP datagrams that have been discarded because they needed to be fragmented at this entity but could not be, e.g., because their Don't Fragment flag was set.
TM_MIB_IFRAGCREATES:	The number of IP datagram fragments that have been generated as a result of fragmentation at this entity.
TM_MIB_IROUTINGDISCARDS:	The number of routing entries which were chosen to be discarded even though they are valid. One possible reason for discarding such an entry could be to free-up buffer space for other routing entries.

**Returns**

**Value**  
NULL  
!=NULL

**Meaning**

The object value was not found.  
Pointer to the found object value.

#### 4.4.6 `tfIpRouteTableEntryGet`

```
long                tfIpRouteTableEntryGet
(
int                exact,
ttMib2IpRouteEntry *ip_route_table
);
```

##### Function Description

This function retrieves the values in a particular row of `ipRouteTable`. The `ttMib2IpRouteEntry` is defined in `trmib2.h` as follows:

```
typedef struct tsMib2IpRouteEntry
{
    unsigned long ipRouteDest;
    unsigned long ipRouteNextHop;
    unsigned long ipRouteMask;
    long ipRouteInfo[TM_SNMP_MAX_NAME_LEN];
    unsigned int ipRouteIfIndex;
    unsigned int ipRouteMetric[5];
    unsigned int ipRouteType;
    unsigned int ipRouteProto;
    unsigned int ipRouteAge;
    unsigned int ipRouteInfoLen;
} ttMib2IpRouteEntry;
```

The rows in `ipRouteTable` are indexed by `ipRouteDest`. On input the calling function must specify this field to indicate the row for which the values of all fields in the `ttMib2IpRouteEntry` to be returned. The calling function must also specify if an exact match is requested by setting the value of `exact` to `TRUE` if so, and to `FALSE` otherwise. This function performs the search of `ipRouteTable` as follows:

- 1) if `exact` is `TRUE`, and the row with the corresponding `ipRouteDest` is found, fill in the remaining fields and return;
- 2) if `exact` is `TRUE`, and the row with the corresponding `ipRouteDest` is not found, return;
- 3) if `exact` is `FALSE`, and the row lexicographically next to the requested one is found, fill in all fields, including `ipRouteDest`, and return;
- 4) if `exact` is `FALSE`, and the row lexicographically next to the requested one is not found, return.

It is the responsibility of this function to sort the rows in ascending order of `ipRouteDest`, if not already sorted. It is the responsibility of a calling function to allocate memory for `ip_route_table`.

##### Parameters

Parameter	Description
<i>exact</i>	The flag indicating whether an exact match is requested ( <code>TRUE</code> or <code>FALSE</code> )
<i>ip_route_table</i>	Pointer to <code>ttMib2IpRouteEntry</code> with the <code>ipRouteDest</code> field filled in on input; pointer to <code>ttMib2IpRouteEntry</code> with all members filled on output

##### Returns

Value	Meaning
0	<i>exact</i> = <code>TRUE</code> and the row was not found, OR <i>exact</i> = <code>FALSE</code> and the row lexicographically following the requested one was not found.
1	<i>exact</i> = <code>TRUE</code> and the row was found, OR <i>exact</i> = <code>FALSE</code> and the row lexicographically following the requested one was found..

#### 4.4.7 tfN2mTableEntryGet

```
long                tfN2mTableEntryGet
(
int                exact,
ttMib2IpNetToMediaEntry *n2m_table
);
```

##### Function Description

This function retrieves the values in a particular row of ipNetToMediaTable. The ttMib2IpNetToMediaEntry is defined in trmib2.h as follows:

```
typedef struct tsMib2IpNetToMediaEntry
{
    unsigned long n2mNetAddr;
    unsigned int n2mIfIndex;
    unsigned int n2mType;
    unsigned char n2mPhysAddr[6];
    char n2mPad[2];
} ttMib2IpNetToMediaEntry;
```

The rows in ipNetToMediaTable are indexed by n2mIfIndex and n2mNetAddr (in that order). On input the calling function must specify these fields to indicate the row for which the values of all fields in the ttMib2IpNetToMediaEntry to be returned. The calling function must also specify if an exact match is requested by setting the value of exact to TRUE if so, and to FALSE otherwise. This function performs the search of ipNetToMediaTable as follows:

- 1) if exact is TRUE, and the row with the corresponding { n2mIfIndex, n2mNetAddr } is found, fill in the remaining fields and return;
- 2) if exact is TRUE, and the row with the corresponding { n2mIfIndex, n2mNetAddr } is not found, return;
- 3) if exact is FALSE, and the row lexicographically next to the requested one is found, fill in all fields, including { n2mIfIndex, n2mNetAddr }, and return;
- 4) if exact is FALSE, and the row lexicographically next to the requested one is not found, return.

It is the responsibility of this function to sort the rows in ascending order of { n2mIfIndex, n2mNetAddr }, if not already sorted. It is the responsibility of a calling function to allocate memory for n2m\_table.

##### Parameters

Parameter	Description
<i>exact</i>	The flag indicating whether an exact match is requested (TRUE or FALSE)
<i>n2m_table</i>	Pointer to ttMib2IpNetToMediaEntry with the { n2mIfIndex, n2mNetAddr } field filled in on input; pointer to ttMib2IpNetToMediaEntry with all members filled on output

##### Returns

Value	Meaning
0	<i>exact</i> = TRUE and the row was not found, OR <i>exact</i> = FALSE and the row lexicographically following the requested one was not found.
1	<i>exact</i> = TRUE and the row was found, OR <i>exact</i> = FALSE and the row lexicographically following the requested one was found.

**Example**

Suppose ipNetToMediaTable contains the following entries (only index, net\_addr are shown):

<b>index</b>	<b>net_addr</b>
1	10.1.23.5
1	128.97.88.44
2	12.97.88.44
2	192.3.11.117
3	202.11.117

If this function is called with `exact = TRUE` and the { `n2mIfIndex`, `n2mNetAddr` } fields (2, 192.3.11.117), the values from the fourth row are filled in, and 1 is returned.

If this function is called with `exact = TRUE` and the { `n2mIfIndex`, `n2mNetAddr` } fields {2, 128.97.88.44}, 0 is returned.

If this function is called with `exact = FALSE` and the { `n2mIfIndex`, `n2mNetAddr` } fields (0, 0.0.0.0), all values from the first row are filled in, and 1 is returned.

If this function is called with `exact = FALSE` and the { `n2mIfIndex`, `n2mNetAddr` } fields {2, 193.0.0.0}, all values from the last row are filled in, and 1 is returned.

If this function is called with `exact = FALSE` and the { `n2mIfIndex`, `n2mNetAddr` } fields {4, 0.0.0.0}, 0 is returned.

#### 4.4.8 `tfTcpGroupInfoGet`

```
long                *tfTcpGroupInfoGet
(
unsigned char      what
);
```

##### Function Description

This function retrieves a value of an object in tcp group. The calling function must specify which object by using the “what” argument.

##### Parameters

Parameter	Description
<i>what</i>	Magic number identifying a member of MIB-II tcp group, one of the following defined in <code>trconsts.h</code> :
<b>what parameter description</b>	
TM_MIB_TCPRTOTALGORITHM	The algorithm used to determine the timeout value used for retransmitting unacknowledged octets.
TM_MIB_TCPRTOMIN:	The minimum value permitted by a TCP implementation for the retransmission timeout, measured in milliseconds. More refined semantics for objects of this type depend upon the algorithm used to determine the retransmission timeout. In particular, when the timeout algorithm is <code>rsre(3)</code> , an object of this type has the semantics of the <code>LBOUND</code> quantity described in RFC 793.
TM_MIB_TCPRTOMAX:	The maximum value permitted by a TCP implementation for the retransmission timeout, measured in milliseconds. More refined semantics for objects of this type depend upon the algorithm used to determine the retransmission timeout. In particular, when the timeout algorithm is <code>rsre(3)</code> , an object of this type has the semantics of the <code>UBOUND</code> quantity described in RFC 793.
TM_MIB_TCPMAXCONN:	The limit on the total number of TCP connections the entity can support. In entities where the maximum number of connections is dynamic, this object should contain the value -1.
TM_MIB_TCPACTIVEOPENS:	The number of times TCP connections have made a direct transition to the <code>SYN-SENT</code> state from the <code>CLOSED</code> state.
TM_MIB_TCPPASSIVEOPENS:	The number of times TCP connections have made a direct transition to the <code>SYN-RCVD</code> state from the <code>LISTEN</code> state.
TM_MIB_TCPATTEMPTFAILS:	The number of times TCP connections have made a direct transition to the <code>CLOSED</code> state from either the <code>SYN-SENT</code> state or the <code>SYN-RCVD</code> state, plus the number of times TCP connections have made a direct transition to the <code>LISTEN</code> state from the <code>SYN-RCVD</code> state.
TM_MIB_TCPSTABRESETS:	The number of times TCP connections have made a direct transition to the <code>CLOSED</code> state from either the <code>ESTABLISHED</code> state or the <code>CLOSE-WAIT</code> state.
TM_MIB_TPCURRESTAB:	The number of TCP connections for which the current state is either <code>ESTABLISHED</code> or <code>CLOSE-WAIT</code> .
TM_MIB_TCPINSEGS:	The total number of segments received, including those received in error. This count includes segments received on currently established connections.
TM_MIB_TCPOUTSEGS:	The total number of segments sent, including those on current connections but excluding those containing only retransmitted octets.
TM_MIB_TCPRETRANSSEGS:	The total number of segments retransmitted – that is, the number of TCP segments transmitted containing one or more previously transmitted octets.
TM_MIB_TCPINERRS:	The total number of segments received in error (e.g., bad TCP checksums).
TM_MIB_TCPOUTRSTS:	The number of TCP segments sent containing the <code>RST</code> flag.

**Returns****Value**

NULL

!=NULL

**Meaning**

The object value was not found.

Pointer to the found object value.

#### 4.4.9 ttTcpTableEntryGet

```
long                ttTcpTableEntryGet
(
int                exact,
ttMib2TcpConnection *tcp_table
);
```

##### Function Description

This function retrieves the values in a particular row of tcpConnTable. The ttMib2TcpConnection is defined in trmib2.h as follows:

```
typedef struct tsMib2TcpConnection
{
    unsigned long tcpConnLocalAddr;
    unsigned long tcpConnRemoteAddr;
    unsigned int tcpConnState;
    unsigned int tcpConnLocalPort;
    unsigned int tcpConnRemotePort;
} ttMib2TcpConnection;
```

The rows in tcpConnTable are indexed by tcpConnLocalAddr, tcpConnLocalPort, tcpConnRemoteAddr and tcpConnRemotePort (in that order). On input the calling function must specify these fields to indicate the row for which the values of all fields in the ttMib2TcpConnection to be returned. The calling function must also specify if an exact match is requested by setting the value of exact to TRUE if so, and to FALSE otherwise. This function performs the search of tcpConnTable as follows:

- 1) if exact is TRUE, and the row with the corresponding { tcpConnLocalAddr, tcpConnLocalPort, tcpConnRemoteAddr, tcpConnRemotePort } is found, fill in the remaining fields and return;
- 2) if exact is TRUE, and the row with the corresponding { tcpConnLocalAddr, tcpConnLocalPort, tcpConnRemoteAddr, tcpConnRemotePort } is not found, return;
- 3) if exact is FALSE, and the row lexicographically next to the requested one is found, fill in all fields, including { tcpConnLocalAddr, tcpConnLocalPort, tcpConnRemoteAddr, tcpConnRemotePort }, and return;
- 4) if exact is FALSE, and the row lexicographically next to the requested one is not found, return.

It is the responsibility of this function to sort the rows in ascending order of { tcpConnLocalAddr, tcpConnLocalPort, tcpConnRemoteAddr, tcpConnRemotePort }, if not already sorted. It is the responsibility of a calling function to allocate memory for tcp\_table.

##### Parameters

Parameter	Description
<i>exact</i>	The flag indicating whether an exact match is requested (TRUE or FALSE)
<i>tcp_table</i>	Pointer to ttMib2TcpConnection with the tcpConnLocalAddr, tcpConnLocalPort, tcpConnRemoteAddr, tcpConnRemotePort fields filled in on input; pointer to ttMib2TcpConnection with all members filled on output

##### Returns

Value	Meaning
0	<i>exact</i> = TRUE and the row was not found, OR <i>exact</i> = FALSE and the row lexicographically following the requested one was not found.
1	<i>exact</i> = TRUE and the row was found, OR <i>exact</i> = FALSE and the row lexicographically following the requested one was found.

**Example**

Suppose `tcpConnTable` contains the following entries (only `tcpConnLocalAddr`, `tcpConnLocalPort`, `tcpConnRemoteAddr`, `tcpConnRemotePort` are shown):

<code>tcpConnLocalAddr</code>	<code>tcpConnLocalPort</code>	<code>tcpConnRemoteAddr</code>	<code>tcpConnRemotePort</code>
10.1.23.5	21	128.11.12.13	23
10.1.23.5	23	18.11.12.13	19
10.1.23.5	23	128.11.12.13	18
128.97.88.44	1234	202.1.1.1	5555
128.97.88.44	1234	202.1.1.1	15001
128.97.88.44	1234	211.10.10.10	23
128.97.88.44	1340	11.10.10.10	22

If this function is called with `exact = TRUE` and the { `tcpConnLocalAddr`, `tcpConnLocalPort`, `tcpConnRemoteAddr`, `tcpConnRemotePort` } fields {128.97.88.44, 1234, 202.1.1.1, 15001}, the values from the fifth row are filled in, and 1 is returned.

If this function is called with `exact = TRUE` and the { `tcpConnLocalAddr`, `tcpConnLocalPort`, `tcpConnRemoteAddr`, `tcpConnRemotePort` } fields {128.97.88.44, 1234, 202.1.1.1, 15000}, 0 is returned.

If this function is called with `exact = FALSE` and the { `tcpConnLocalAddr`, `tcpConnLocalPort`, `tcpConnRemoteAddr`, `tcpConnRemotePort` } fields (0.0.0.0, 0, 0.0.0.0, 0), all values from the first row are filled in, and 1 is returned.

If this function is called with `exact = FALSE` and the { `tcpConnLocalAddr`, `tcpConnLocalPort`, `tcpConnRemoteAddr`, `tcpConnRemotePort` } fields {128.97.88.44, 1234, 202.1.1.1, 15002}, all values from the sixth row are filled in, and 1 is returned.

If this function is called with `exact = FALSE` and the { `tcpConnLocalAddr`, `tcpConnLocalPort`, `tcpConnRemoteAddr`, `tcpConnRemotePort` } fields {128.97.88.44, 1340, 202.1.1.1, 15000}, 0 is returned.

#### 4.4.10 `tfUdpGroupInfoGet`

```
long                *tfUdpGroupInfoGet
(
unsigned char      what
);
```

##### Function Description

This function retrieves a value of a statistics counters in udp group. The calling function must specify which statistics counter by using the “what” argument.

##### Parameters

Parameter	Description
-----------	-------------

*what*

Magic number identifying a member of MIB-II udp group, one of the following defined in `trconsts.h`:

##### *what* parameter descriptions

TM\_MIB\_UDPINDATAGRAMS:

The total number of UDP datagrams delivered to UDP users.

TM\_MIB\_UDPNOPORTS:

The total number of received UDP datagrams for which there was no application at the destination port.

TM\_MIB\_UDPINERRORS:

The number of received UDP datagrams that could not be delivered for reasons other than the lack of an application at the destination port.

TM\_MIB\_UDPOUTDATAGRAMS:

The total number of UDP datagrams sent from this entity.

##### Returns

Value	Meaning
-------	---------

NULL

The statistics counter value was not found.

!=NULL

Pointer to the found statistics counter value.

#### 4.4.11 `tfUdpTableEntryGet`

```
long          tfUdpTableEntryGet
(
int          exact,
ttMib2UdpListener *udp_table
);
```

##### Function Description

This function retrieves the values in a particular row of `udpTable`. The `ttMib2UdpListener` is defined in `trmib2.h` as follows:

```
typedef struct tsMib2UdpListener
{
    unsigned long udpLocalAddr;
    unsigned int  udpLocalPort;
} ttMib2UdpListener;
```

The rows in `udpTable` are indexed by `udpLocalAddr` and `udpLocalPort` (in that order). On input the calling function must specify these fields to indicate the row for which the values of all fields in the `ttMib2UdpListener` to be returned. The calling function must also specify if an exact match is requested by setting the value of `exact` to `TRUE` if so, and to `FALSE` otherwise. This function performs the search of `udpTable` as follows:

1. if `exact` is `TRUE`, and the row with the corresponding { `udpLocalAddr`, `udpLocalPort` } is found, fill in the remaining fields and return;
2. if `exact` is `TRUE`, and the row with the corresponding { `udpLocalAddr`, `udpLocalPort` } is not found, return;
3. if `exact` is `FALSE`, and the row lexicographically next to the requested one is found, fill in all fields, including { `udpLocalAddr`, `udpLocalPort` }, and return;
4. if `exact` is `FALSE`, and the row lexicographically next to the requested one is not found, return.

It is the responsibility of this function to sort the rows in ascending order of { `udpLocalAddr`, `udpLocalPort` }, if not already sorted. It is the responsibility of a calling function to allocate memory for `udp_table`.

##### Parameters

Parameter	Description
<i>exact</i>	The flag indicating whether an exact match is requested ( <code>TRUE</code> or <code>FALSE</code> )
<i>udp_table</i>	Pointer to <code>ttMib2UdpListener</code> with the <code>udpLocalAddr</code> , <code>udpLocalPort</code> fields filled in on input; pointer to <code>ttMib2UdpListener</code> with all members filled on output

##### Return

Value	Meaning
0	<i>exact</i> = <code>TRUE</code> and the row was not found, OR <i>exact</i> = <code>FALSE</code> and the row lexicographically following the requested one was not found.
1	<i>exact</i> = <code>TRUE</code> and the row was found, OR <i>exact</i> = <code>FALSE</code> and the row lexicographically following the requested one was found.

**Example**

Suppose `udpTable` contains the following entries:

<b>udpLocalAddr</b>	<b>udpLocalPort</b>
10.1.23.5	161
10.1.23.5	1000
128.97.88.44	17
128.97.88.44	1401

If this function is called with `exact = TRUE` and the { `udpLocalAddr`, `udpLocalPort` } fields (128.97.88.44, 17), 1 is returned.

If this function is called with `exact = TRUE` and the { `udpLocalAddr`, `udpLocalPort` } fields {128.97.88.44, 18}, 0 is returned.

If this function is called with `exact = FALSE` and the { `udpLocalAddr`, `udpLocalPort` } fields (0, 0.0.0.0), the values from the first row are filled in, and 1 is returned.

If this function is called with `exact = FALSE` and the { `udpLocalAddr`, `udpLocalPort` } fields {128.0.0.0, 0}, the values from the third row are filled in, and 1 is returned.

If this function is called with `exact = FALSE` and the { `udpLocalAddr`, `udpLocalPort` } fields {128.97.88.45, 0}, 0 is returned.

#### 4.4.12 `tfWriteAdminStat`

```
int          tfWriteAdminStat
(
int          row,
long        value
);
```

##### Function Description

This function is called to modify `ifAdminStatus`, the object in the interfaces group of MIB-II, changing the state of the interface. `ifAdminStatus` can take on values: `up` (1), `down` (2), and `testing` (3). The testing state indicates that no operational packets can be passed through this interface.

##### Parameters

Parameter	Description
<i>row</i>	The 1-based interface index.
<i>value</i>	up (1), or down (2), or testing (3)

##### Returns

Value	Meaning
<code>TM_SNMP_ERR_NOERR</code>	Success.
<code>TM_SNMP_ERR_GENERR</code>	Non-specific error occurred.

#### 4.4.13 tfWriteIp

```
int          tfWriteIp
(
int          what,
long        value
);
```

##### Function Description

This function is called to modify ipForwarding or ipDefaultTTL, the objects in the ip group of MIB-II, changing the ip parameters of the entity. ipForwarding allows the entry to act or not to act as a gateway, with the values forwarding (1) and not-forwarding (2), correspondingly. The ipDefaultTTL is inserted into the Time-To-Live field of the IP header of datagrams originated at this entity

##### Parameters

Parameter	Description
-----------	-------------

<i>what</i>	1, for ipForwarding; 2 for ipDefaultTTL
-------------	---

<i>value</i>	forwarding (1), or not-forwarding (2) for ipForwarding; integer value for ipDefaultTTL
--------------	--

##### Returns

Value	Meaning
-------	---------

TM_SNMP_ERR_NOERR	Success.
-------------------	----------

#### 4.4.14 tfWriteIpRouteEntry

```
int          tfWriteIpRouteEntry
(
long        address,
int         what,
long        value
);
```

##### Function Description

This function is called to modify every row member in ipRouteTable, except ipRouteProto and ipRouteInfo. The modifiable objects are: ipRouteDest, ipRouteIfIndex, ipRouteMetric1, ipRouteMetric2, ipRouteMetric3, ipRouteMetric4, ipRouteNextHop, ipRouteType, ipRouteAge, ipRouteMask and ipRouteMetric5. The ipRouteTable is indexed by ipRouteDest, and the argument *address* indicates ipRouteTable row, one of whose members is to be modified. The argument *what* denoted the member of the row whose instance is to be modified, and *value* contains the new instance. NOTE: the table index itself, ipRouteDest, can be modified. In this case a special care should be exercised: first, delete the existing row saving the values of all members in that row, then add another row with members assuming these values and the new ipRouteDest equal to *value*.

##### Parameters

###### Parameter

*address*

*what*

*value*

###### Description

ipRouteDest address of the row to be modified

A member of the row whose instance is to be modified. The following values are defined:

- 1 = ipRouteDest
- 2 = ipRouteIfIndex
- 3 = ipRouteMetric1
- 4 = ipRouteMetric2
- 5 = ipRouteMetric3
- 6 = ipRouteMetric4
- 7 = ipRouteNextHop
- 8 = ipRouteType
- 10 = ipRouteAge
- 11 = ipRouteMask
- 12 = ipRouteMetric5

New numeric value for the member in the row

##### Returns

###### Value

TM\_SNMP\_ERR\_NOERROR

TM\_SNMP\_ERR\_NOSUCHNAME

TM\_SNMP\_ERR\_NOTWRITABLE

TM\_SNMP\_ERR\_GENERR

###### Meaning

Success.

Failure to find the requested row.

Field specified by *what* is not writable.

Non-specific error occurred.

##### Example

Assume one of ipRouteTable rows contains 0x0a010101 (10.1.1.1) as ipRouteDest value. Then if the call is made tfWriteIpRouteEntry(0x0a010101, 10, 0xffff0000), the function should change ipRouteMask to 255.255.0.0 and return. But if the function is called as tfWriteIpRouteEntry(0x0a010101, 1, 0x0a010102), then the entire row should be saved and consequently deleted, and a new row should be created with ipRouteDest value 0x0a010102 (10.1.1.2).

#### 4.4.15 tfWriteN2mEntry

```
long                tfWriteN2mEntry
(
ttMib2IpNetToMediaEntry *n2m_old_table,
ttMib2IpNetToMediaEntry *n2m_new_table
);
```

##### Function Description

This function is called to modify the row entry in ipNetToMediaTable. The modifiable objects are members of the ttMib2IpNetToMediaEntry defined in trmib2.h as follows:

```
typedef struct tsMib2IpNetToMediaEntry
{
    unsigned long n2mNetAddr;
    unsigned int n2mIfIndex;
    unsigned int n2mType;
    unsigned char n2mPhysAddr[6];
    char n2mPad[2];
} ttMib2IpNetToMediaEntry;
```

The ipNetToMediaTable is indexed by ipNetToMediaIndex and ipNetToMediaNetAddress ({n2mIfIndex, n2mNetAddr }). The calling function passes as the first argument, n2m\_old\_table, the pointer to ttMib2IpNetToMediaEntry with index and net\_addr fields filled in and identifying the valid row entry in ipNetToMediaTable. The second argument, n2m\_new\_table, is also a pointer to ttMib2IpNetToMediaEntry with one field having a non-zero value, and the remaining fields set to 0. The field having the non-zero value is that field that is being modified, and this function is called to modify only a single field at a time. If one of n2mPhysAddr or n2mType is non-zero, this function must modify the field values accordingly and return. If one of n2mIfIndex or n2mNetAddr is non-zero, the function must delete the existing row saving the values of all members not subject to change, and create a new row with the new index field value specified. It is the responsibility of a calling function to allocate memory for n2m\_old\_table and n2m\_new\_table.

##### Parameters

Parameter	Description
<i>n2m_old_table</i>	pointer to ttMib2IpNetToMediaEntry with the index { n2mIfIndex, n2mNetAddr } filled in on input
<i>n2m_new_table</i>	pointer to ttMib2IpNetToMediaEntry with the values filled in on input

##### Returns

Value	Meaning
TM_SNMP_ERR_NOERROR	Success.
TM_SNMP_ERR_NOSUCHNAME	Failure to find the requested row.
TM_SNMP_ERR_GENERR	Non-specific error occurred.

**Example**

Assume one of `ipNetToMediaTable` rows contains the following values:

```

IpNetToMediaIfIndex      2
IpNetToMediaPhysAddress  0x080020010203
IpNetToMediaNetAddress  192.168.1.2
IpNetToMediaType         4

```

Then if the arguments in the calling function `tfWriteN2mEntry` point to `n2m_old_table` with the values:

```

n2mIfIndex      2
n2mPhysAddr     0x000000000000
n2mNetAddr      192.168.1.2
n2mType         0

```

and `n2m_new_table` with the values:

```

n2mIfIndex      0
n2mPhysAddr     0x0a0010010203
n2mNetAddr      0
n2mType         0

```

then `IpNetToMediaPhysAddress` in the existing entry is modified from `0x080020010203` to `0x0a0010010203`, and the function returns.

If, on the other hand, the second argument of `tfWriteN2mEntry` points to `n2m_new_table` with the values:

```

n2mIfIndex      1
n2mPhysAddr     0x000000000000
n2mNetAddr      0
n2mType         0

```

then this function must save the value of `n2mNetAddr` (192.168.1.2) and `n2mPhysAddr` (0x080020010203), and delete the row. Next, the function must create a new row with the values:

```

IpNetToMediaIfIndex      1
IpNetToMediaPhysAddress  0x080020010203 (saved)
IpNetToMediaNetAddress  192.168.1.2 (saved)
IpNetToMediaType         4 (saved) and return.

```

#### 4.4.16 tfWriteTcpState

```
long                tfWriteTcpState
(
ttMib2TcpConnection *tcp_conn_entry
);
```

##### Function Description

This function is called to modify tcpConnState, the object in tcpConnTable. The calling function passes as the argument the pointer to defined in trmib2.h:

```
typedef struct tsMib2TcpConnection
{
    unsigned long tcpConnLocalAddr;
    unsigned long tcpConnRemoteAddr;
    unsigned int tcpConnState;
    unsigned int tcpConnLocalPort;
    unsigned int tcpConnRemotePort;
} ttMib2TcpConnection;
```

The values of tcpConnLocalAddr, tcpConnLocalPort, tcpConnRemoteAddr and tcpConnRemotePort are filled in on input to identify the entry in tcpConnTable. The value of tcpConnState can be set only to deleteTCB (12). Accordingly, the value of the member tcpConnState is set to 12. There is no need to verify this value in this function since the calling function has already done so. This function just modifies the value of tcpConnState and returns. It is the responsibility of a calling function to allocate memory for tcp\_conn\_entry.

##### Parameters

Parameter	Description
tcp_conn_entry	pointer to ttMib2TcpConnection with the index { tcpConnLocalAddr, tcpConnLocalPort, tcpConnRemoteAddr, tcpConnRemotePort } and tcpConnState value filled in on input

##### Returns

Value	Meaning
TM_SNMP_ERR_NOERROR	Success.
TM_SNMP_ERR_NOSUCHNAME	Failure to find the requested row.
TM_SNMP_ERR_GENERR	Non-specific error occurred.

```
/**      Insert your code here      */
return 0;
}

int rip2ifconftable_entry_get(int exact,
    struct rip2ifconftable_entry *entry0)

/*
 * exact - 1 if the exact value is retrieved; 0 otherwise
 * entry0 - pointer to the struct rip2ifconftable_entry
 * RETURN: 1 on success, 0 on failure
 */
{
/*
 * This function is called with the pointer to the
 * struct rip2ifconftable_entry
 * On entry the indices are set, to allow to identify the
 * row, and all other fields are bzero'd. The index (indices)
 * may or may not be valid row index (indices).
 * If exact == 1 and the row is found, fill in ALL fields in
 * a given entry and return 1.
 * If exact == 1 and the row is not found, return 0.
```