SNMP
User's Guide

Version 2.58
July 2004

U S SOFTWARE®
EMBEDDED EXCELLENCE

Copyright and Trademark Information

Lantronix, Inc.
15353 Barranca Parkway
Irvine, CA 92618
(949)453-3990
Fax (949) 453-3995

For Support Contact:
Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxinfo.com
www.smxinfo.com

Documentation Conventions

Computer output and code examples: `Courier`, usually in a separate paragraph.

Function names and command names: ***Bold italic***, usually followed by parentheses, as in ***main()*** function.

Variables: Courier 11 italic (*`mt_busy`*).

File names: Times bold (the file **usrclk.asm**), in lower case.

Key names: Initial capital, in angle brackets, as in press <Enter>.

Menu names and selections, dialog box names, screen titles, window titles: Times bold, as in **File** menu**.**

Notes: Indicate important information.

Cautions: Indicate potential damage to hardware or data.

Documentation History

Revision Number	Date
2.1	October l996
2.5	August 1997
2.53	May 1998
2.57	July 2000
2.58	October 2001

Contents

Introduction

This manual describes the use of the Simple Network Management Protocol for USNET®. SNMP/USNET provides support for integrating an SNMP agent into a real-time embedded system application. It is designed for use with SNMP Version 3 managers; however, it will also respond to Version 1 and 2 requests.

The reader ought to have a conceptual knowledge of SNMP in order to understand the terminology in this manual. There are several books available that explain more completely the terminology and function of SNMP systems.

Recommended Reading

USNET User's Manual

The Simple Book
An Introduction to Internet Management
Second Edition
Marshall T. Rose
ISBN 0-13-177254-6

SNMP, SNMPv2, SNMPv3, and RMON 1 and 2
Practical Network Management
William Stallings
ISBN 0-201-48534-6

SNMP Overview

The Simple Network Management Protocol, SNMP, is used widely by industry to manage networks. On a network, a client in one host (a SNMP manager) communicates with a server in another host (an SNMP agent). The manager requests the agent to read or write information (objects) in a Management Information Base (MIB).

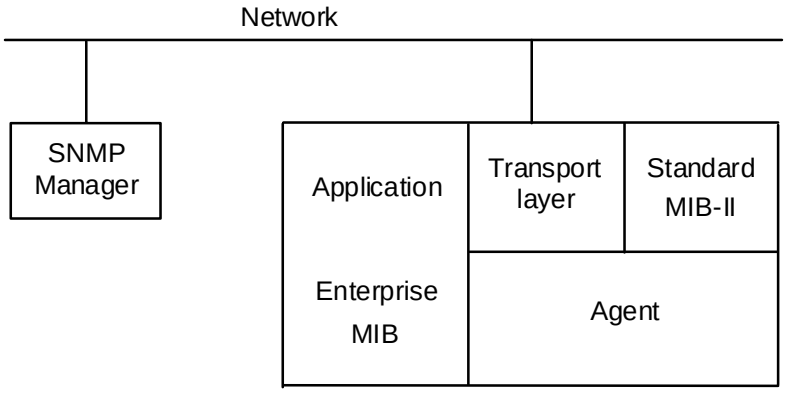

Figure 1: SNMP Agent on a network

Design of SNMP/USNET

The SNMP/USNET design includes these features:

- Independent of processor, operating system, and networking stack
- ROMable
- Compact
- User-configurable

The agent is processor-independent. Almost any ANSI C compiler will do.

The agent is not tied to a particular operating system. It can be executed as a task in nearly any commercial or custom multitasking operating system, as a polling routine in a single-threaded application, or as the entire application.

The agent is not tied to a particular transport layer. Any networking stack or other data communication layer can be used to transfer data to and from the agent. Sample support for Linux*, Windows, and USNET, each using UDP/IP, is provided.

The code is ROMable in that all initialized data is type `const`, and there are no attempts to change code or constants at runtime.

The agent requires less than 30K code bytes and 12K RAM bytes on a typical compiler without optimization. If security is removed, the agent requires less than 20K code bytes and 4K RAM bytes. Actual code requirements also vary somewhat from processor to processor and compiler to compiler.

SNMP/USNET supports the same application interface and functionality across all processors. In other words, standard C code developed for one processor can be recompiled for another processor with minimal effort.

Custom MIBs can be created using the MIB compiler supplied with the SNMP/USNET agent. The application can add these new MIBs or remove old or unused MIBs with relative ease.

Building an Application

Build-time Configuration

The build-time configuration of the agent is performed in the **snmpcfg** and **vacm.c** files in the **snmpsrc** directory. In the file **snmpcfg** there is a set of definitions used to configure the agent. These symbolic constants may require modification before compiling and linking the product. The View-based Access Control Model definitions are declared in **vacm.c**.

Constants

ENTERPRISE Constant

The `ENTERPRISE` value refers to the `ENTERPRISE ID` assigned by ICANN (formerly IANA). It is used to partly form the *`snmpEngineID`* for the agent.

```
ENTERPRISE = 991
```

SNMP_IP Constant

The `SNMP_IP` value is the IP address of the host. It is most convenient to determine this at run time if the host address may change or is unknown. However, the *`snmpEngineID`* is partly determined from the IP address of the agent, and the User-based Security Model (USM) requires that an intensive algorithm using the *`snmpEngineID`* be performed on each password used. There is more information about this in the section on configuring the USM.

```
SNMP_IP = “198.107.35.105”
```

System Variable Constants

The MIB system group used by the agent provides a textual description of the agent and is required by SNMP. These strings can be modified, adding appropriate values for the particular agent application. These variables are shown in Table 1.

Table 1: System Variables

Variable	Description
SYSCONTACT	The value is stored in *system.sysContact*. Replace this value with the company name and phone number.
SYSLOCATION	The value is stored in *system.sysLocation*. Replace this with the company name and address.
SYSDESCR	The value is stored in *system.sysDescr*. Replace this string with a description for the agent

Note that these values should not be greater than 64 bytes each without changing the size of the arrays that hold them. See *sysContact*, *sysLocation*, and *sysDescr* in **snmpsrc\v3\agent.c**.

```
SYSCONTACT = "USSW (503) 844-6614, support@ussw.com"

SYSLOCATION = "USSW  Hillsboro, OR  USA"

SYSDESCR = "Embedded controller running USNET"
```

ENABLEAUTHENTRAPSVAL Constant

The ENABLEAUTHENTRAPSVAL specifies the *snmpEnableAuthentTrapsVal* default value. Use 1 for enabled and 2 for disabled.

```
ENABLEAUTHENTRAPSVAL = 2
```

MAXOID Constant

MAXOID defines the maximum length of an object identifier in the MIB. The object identifier (OID) uniquely defines MIB variables. Be sure this is large enough to accommodate all objects within any application MIB.

MAXOID Example

```
#define MAXOID 12  /* maximum length of object ID */
static const struct
  {unsigned char nlen, name[MAXOID], key[16];}
  party[]={
    {11, {0x2b,6,1,6,3,3,1,3,10,11,12}, {0} },
    {11, {0x2b,6,1,6,3,3,1,3,10,11,13}, {0} },
    {11, {0x2b,6,1,6,3,3,1,3,10,11,14},
    {0x74,0x68,0x69,0x73,0x74,0x68,0x69,
     0x73,0x74,0x68,0x69,0x73,0x74,0x68,0x69,0x33} },
};
```

The `name` field in the above table stores SNMP object IDs, and `MAXOID` specifies the maximum size for this value. Note that the OIDs start with the value `0x2b`, which is the BER encoding for .1.3.

```
MAXOID = 15
```

MAXKEY Constant

`MAXKEY` defines the maximum number of keys allowed. Keys form the index used to identify a MIB table entry. For example, the *`tcpConnTable`* has four keys: *`tcpConnLocalAddress`*, *`tcpConnLocalPort`*, *`tcpConnRemAddress`*, and *`tcpConnRemPort`*. No other table in the MIB-II has more than four, so `MAXKEY` can be set to 4.

```
MAXKEY = 4
```

MAXKLEN Constant

`MAXKLEN` defines the maximum length in bytes for an encoded index. An index is the encoding of the keys used to define a table entry. These keys may be one or more of nearly any fixed length data type such as *`IpAddress`* or *`INTEGER`*. For standard MIB-II objects, the largest possible index is potentially generated by the *`tcpConnTable`*. Its keys include two *`IpAddresses`* each up to 8 bytes encoded and two 16-bit unsigned integers each up to 3 bytes encoded. The result is 22 bytes.

```
MAXKLEN = 22
```

MAXVAR Constant

MAXVAR specifies the maximum number of variables allowed in a request. A request is a message sent by the manager to the agent for reading or setting values of one or more variables. MAXVAR sets the maximum number of variables that may be accessed in one request. Note that the number of total response variables for a response to a bulk request is limited by the packet size, not this constant.

```
MAXVAR = 16
```

SNMP_MAXSIZE Constant

SNMP_MAXSIZE specifies the maximum transport size in bytes. Note that this value represents the size of each of four SNMP message buffers used for the following purposes: Receiving requests, sending replies, sending traps, and performing security operations. RFC 2571 requires this value be at least 484 bytes.

```
SNMP_MAXSIZE = 1000
```

User-based Security Model Configuration

The current agent supports *authPriv* (i.e. authentication with privacy), *authNoPriv* (i.e. authentication without privacy), *noAuthNoPriv* (i.e. no authentication and no privacy) for security levels.

SNMPv3 defines a method of security known as the User-based Security Model (USM). The definition in RFC 2574 encompasses both authentication and privacy. Authentication means the verification of host identity, usually through a user name and password. Privacy means the encryption of SNMP messages such that unauthorized hosts cannot interpret the data. The current agent supports *authNoPriv* (i.e. authentication without privacy) and *noAuthNoPriv* (i.e. no authentication and no privacy) for security levels. Future versions may add new authentication and privacy protocols.

By default, the agent comes with two users defined. The first uses the *noAuthNoPriv* security level. The second uses the *authNoPriv* security level. These are configured in **snmpsrc\v3\snmpcfg** with the following definition:

```
# # # #
#
# The entries below define system users.  They use the
# following format:
#
#   “usmUserSecurityName”
#   usmUserAuthProtocol “auth-pass-phrase” \
#   usmUserPrivProtocol “priv-pass-phrase”
#
#
# The possible choices for usmUserAuthProtocol are:
#   usmNoAuthProtocol             No authentication
#   usmHMACMD5AuthProtocol  HMAC MD5 authentication
#   usmHMACSHAAuthProtocol  HMAC SHA authentication
#
# The possible choices for usmUserPrivProtocol are:
#   usmNoPrivProtocol       No privacy
#   usmDESPrivProtocol      DES CBC encryption
#
```

```
U1 = “initial”
   usmNoAuthProtocol “” \
   usmNoPrivProtocol “”
U2 = “admin”
   usmHMACMD5AuthProtocol “secretpassword” \
   usmNoPrivProtocol “mylittlesecret”

U3 = "admin-sha" \
   usmHMACSHAAuthProtocol "mylittlesecret" \
   usmDESPrivProtocol "secretpassword"
USM_ENTRIES = U1 U2 U3
```

The ‘U1’ entry defines an unauthenticated user with the name “initial” and no password. The ‘U2’ entry defines an authenticated user with the name “admin-md5” and the password “secretpassword” using HMAC-MD5 authentication. The 'U3' entry defines an authenticated user with the name "admin-sha" and the password "mylittlesecret" using HMAC-SHA authentication. Note that either entry U2 or U3 may also use DES CBC encryption for privacy with the respective passphrases "mylittlesecret" and "secretpassphrase".

It would not be secure to transmit passwords over the network, so the authors of SNMPv3 came up with a scheme to hide passwords. This method is called password localization and is described in RFC 2574 in section A.2. It takes the password and the *snmpEngineID* as input and outputs a digest-specific key. A SNMP manager uses the key with each SNMP request message to form an authentication digest using HMAC-MD5 or HMAC-SHA, and transmits the message plus the new digest as an authenticated SNMP message. The agent checks each digest value with the digest it creates in the same fashion on each message. If the two match, the management station and agent must have used the same localized password for the request to be further processed. Otherwise, the request causes the agent to transmit a *usmStatsWrongDigests* report to the manager.

Application Note: The password localization algorithm is intensive enough that a typical embedded processor of today probably cannot handle the process in a timely manner at run time. Therefore, our SNMP agent can optionally have its user passwords localized at build time by the development machine. The problem with this is that the *snmpEngineID* is required to be unique for a given communications context. The SNMP/USNET agent assumes the presence of an IP-oriented network and uses IP addresses as the host identifier in the *snmpEngineID*. IP addresses are often dynamically configured. Therefore, the agent may have difficulty being both unique and timely while supporting USM authentication.

The *snmpEngineID* used by the agent concatenates the *ENTERPRISE* value and the transport layer IP address. The *ENTERPRISE* value must always be configured in **snmpcfg**, but the IP address can be configured at run time or in **snmpcfg**.

View-based Access Control Configuration

SNMPv3 defines a method of access control known as the View-based Access Control Model (VACM). It is defined in RFC 2575 as a means of restricting access to particular subsets of variables based on the identity of the manager and *securityLevel* used in the request.

A view is a group of MIB variables on the agent. The agent defines a view for each user based on the user identity and *securityLevel*. A *contextName* and a *securityName* define the user identity and the *securityLevel* is listed directly in each request. Note that if no security is used (i.e. *securityLevel* == *noAuthNoPriv*), the *securityName* can be undefined. Also, in order to provide compatibility with version 1 and 2c management stations, the *contextName* in each view entry may refer to either a *contextName* or a *community name*. The *securityLevel* would then be assumed to be *noAuthNoPriv*.

The general practice is that informational variables be accessible to all users with all security levels. Write access and read access to sensitive information are limited to selective users implementing authentication and perhaps privacy. Generally, if a user uses greater security than is required by the access entry including a particular variable, access is allowed. The VACM module will search through each entry until it finds a valid entry for the variable. This way multiple entries can be defined for a single *securityName* given different combinations of *contextNames* and *securityLevels*.

The configuration of the View-based Access Control Model (VACM) cannot be performed in **snmpcfg** due to the complexity of the procedure. Instead, an array of ACCESS structures is used to define all entries in the VACM table.

These example entries are defined in the **snmpsrc\vacm.c** module:

```
typedef struct
{
    uint32 mask;
    const OID *oid;
} VIEW;

typedef struct
{
    const uint8 *group;     /* Group for entry */
    const uint8 *context;   /* Context for entry */
    const VIEW *readview;   /* Read view for entry */
    const VIEW *writeview;  /* Write view for entry */
    uint16 level;       /* Security level for entry */
} ACCESS;

/*
** Define all possible view sub-trees:
**  Note that we only define 3. Theoretically, there could be
**  a different sub-tree defined for every single OID in
**  every MIB on the host.  This is of course less efficient
**  than the method below, though probably more secure.  This
**  ought to be changed according to the desires of the MIB
**  implementor.
*/
```

```
static const OID sys_oid = {6, {0x2b, 6, 1, 2, 1, 1}};
static const OID mgmt_oid = {4, {0x2b, 6, 1, 2}};
static const OID snmp_oid = {4, {0x2b, 6, 1, 6}};
/* ENTERPRISE 991 when encoded is 0x87 0x5f */
static const OID private_oid = {7, {0x2b, 6, 1, 4, 1, 0x87, 0x5f}};

/*
** Define all possible view families:
**  Note, these arrays always end in NULL so that they can be
**  searched through without a length value being specified.
*/
static const VIEW sys_view[] =
{
    {0xffffffff, &sys_oid},
    {0, 0}
};

    static const VIEW mib2_view[] =
    {
        {0xffffffff, &mgmt_oid},
        {0xffffffff, &snmp_oid},
        {0, 0}
    };

static const VIEW admin_view[] =
{
    {0xffffffff, &mgmt_oid},
    {0xffffffff, &snmp_oid},
    {0xffffffff, &private_oid},
    {0, 0}
};

/* Define all possible access entries */
static const ACCESS vacmAccessTable[] =
```

```
{
   /*
    ** no group,
    ** public context,
    ** no security,
    ** read mib2_view,
    ** no write view
    */
    {(const uint8 *)"", (const uint8 *)"public", sys_view,
            0, noAuthNoPriv},

   /*
    ** initial group,
    ** public context,
    ** no security,
    ** read mib2_view,
    ** no write view
    */
    {(const uint8 *)"initial", (const uint8 *)"public",
            mib2_view, noAuthNoPriv},
   /*
    ** admin group,
    ** public context,
    ** no security,
    ** read mib2_view,
    ** no write view
    */
    {(const uint8 *)"admin", (const uint8 *)"public",
            mib2_view, noAuthNoPriv},
   /*
    ** admin-md5 group,
    ** admin context,
    ** with auth security,
    ** read admin_view,
    ** write mib2_view
    */
    {(const uint8 *)"admin-md5", (const uint8 *)"admin",
            admin_view, mib2_view, authNoPriv}
};
```

```
/*
    ** admin-sha group,
    ** admin context,
    ** with authPriv security,
    ** read admin_view,
    ** write admin_view
    */
    {(const uint8 *)"admin-sha", (const uint8 *)"admin",
            admin_view, admin_view, authPriv}
    };
    #define ACNUM ((sint16)(sizeof(vacmAccessTable) /
                  sizeof(ACCESS)))
```

Agent Use of Build-time Constants

The constants from the previous section translate into **snmpconf.h** and **usmauto.c** during build-time. Here is the **snmpconf.h** output from the above definitions:

```
/* This file is autogenerated */
#define ENTERPRISE 991
#define SYSCONTACT "USSW (503) 844-6614, support@ussw.com"
#define SYSLOCATION "USSW Hillsboro, OR USA"
#define SYSDESCR "Embedded controller running USNET"
#define ENABLEAUTHENTRAPSVAL 2
#define MAXOID 15
#define MAXKEY 4
#define MAXKLEN 22
#define MAXVAR 16
#define SNMP_MAXSIZE 1000
#define USM_MD5
#define USM_DES
#define USM_SHA
```

Here is the **usmauto.c** output from the above definitions:

```
/* This file is autogenerated */

/* User security entries (securityName, auth-type, priv-type) */
static const USER usmUserTable[] =
{
    {(const uint8 *)”initial”, usmNoAuthProtocol,
      usmNoPrivProtocol},
    {(const uint8 *)"admin-md5", usmHMACMD5AuthProtocol,
      usmDESPrivProtocol},
    {(const uint8 *)”admin-sha”, usmHMACSHAAuthProtocol,
      usmDESPrivProtocol},
};
static const uint8 *authPass[] =
{
    (const uint8 *)””,
    (const uint8 *)”secretpassword”,
    (const uint8 *)”mylittlesecret”,
};
static uint8 authKeys[sizeof(usmUserTable) /
      sizeof(USER)][20];

static const uint8 *privPass[] =
{
    (const uint8 *)””,
    (const uint8 *)”mylittlesecret”,
    (const uint8 *)”secretpassword”,
};
static uint8 privKeys[sizeof(usmUserTable) / sizeof(USER)][20];
```

Application Interface

The application file defines the run-time environment in which the agent executes. The AGENT_CONTEXT structure is used to pass the configuration information from the application to the agent.

AGENT_CONTEXT Structure

```
typedef struct
{
    const MIB **mibs;   /* Array of pointers to host MIBs */
    uint16 nummibs;            /* Number of host MIBs */
    const TRAP_HOST **thosts;  /* Trap hosts */
    uint16 numthosts;          /* Number of trap hosts */
    uint16 trapv, trapt; /* Trap version and startup type */
    const TRANSPORT_MAPPING *tm; /* Transport mapping */
} AGENT_CONTEXT;
```

The `mibs` field is the list of MIBs that managers may have access to. Note it is vital that the MIBs be listed in lexicographical order. If not, the agent will think certain variables do not exist within the MIB. The `nummibs` field specifies the number of MIBs available.

The `thosts` field specifies the hosts to which agent traps will be sent. The `TRAP_HOST` definition is simply '`typedef uint8 *TRAP_HOST;`' and each host should be acceptable to the transport layer. In other words, the transport layer needs to be able to open a connection to the entity specified by the trap host field. The `numthosts` field specifies the number of trap hosts available.

The `trapv` field specifies the trap version to use during agent operations. The `trapt` field specifies the trap used by the agent during startup. Use –1 for none. Otherwise use one of these defined types from **snmpv3.h**:

```
COLDSTART
WARMSTART
LINKDOWN
LINKUP
AUTHENTICATIONFAILURE
```

```
EGPNEIGHBORLOSS
ENTERPRISESPECIFIC
```

The `tm` field specifies the transport mapping to be used by the agent. The `TRANSPORT_MAPPING` data structure is defined later.

Example

This is an example of a SNMP agent application taken from **agv3.c**.

A global structure is declared for the agent task to initialize from. In this example, the structure has been set up to request a SNMPv1 (0) `COLDSTART` trap be sent when the agent is started. The USNET DPI transport mapping is used for sending and receiving SNMP packets.

```
#include “snmpv3.h”

extern const MIB mib_if, mib_at, mib_ip, mib_icmp, mib_tcp, mib_udp;
extern const MIB mib_sys, mib_snmp, mib_engine;
extern const MIB mib_usm;

/* The following MIBs must be in lexicographical order */
static const MIB *mibs[] =
{
    &mib_sys,       /* system group */
    &mib_if,        /* interfaces group */
    &mib_at,        /* address translation group */
    &mib_ip,        /* IP group */
    &mib_icmp,      /* ICMP group */
    &mib_tcp,       /* TCP group */
    &mib_udp,       /* UDP group */
    &mib_snmp,      /* SNMP group */
    &mib_engine,    /* SNMPv3 engine group */
    &mib_usm        /* USM group */
};

static const TRAP_HOST primary = “192.168.1.30”;
static const TRAP_HOST secondary = “192.168.1.31”;
```

```
static const TRAP_HOST *thosts[] =
{
    &primary,
    &secondary
};

extern const TRANSPORT_MAPPING TM_DPI;

/* This structure is defined as external in SNMPAgentTask() */
const AGENT_CONTEXT snmp_ac =
{
    mibs, (sizeof(mibs) / sizeof(MIB *)),
    thosts, (sizeof(thosts) / sizeof(TRAP_HOST)), 0, COLDSTART,
    &TM_DPI
};
. . .
```

ussSNMPAgentTask

Executes the agent as a task.

```
void ussSNMPAgentTask(void);
```

This function calls ***ussSNMPAgentInit()*** with the address of the global constant `AGENT_CONTEXT`, `snmp_ac`, defined by the calling application. It then continuously calls ***ussSNMPAgentCheck()***. If the return value of ***ussSNMPAgentCheck()*** indicates an error condition, a call is made to the ***ussSNMPAgentShut()*** function and this function returns.

Example

This is an example of a SNMP agent application taken from ***agv3.c***.

Before running the agent, the system must initialize. In this case, USNET must be initialized with the ***Ninit()*** and ***Portinit()*** calls because the USNET DPI transport mapping is being used.

ussSNMPAgentTask() is called to start the agent. If all goes well, the agent task will remain in a loop waiting for and responding to SNMP manager requests.

```
#include “net.h”
#include “local.h”
#include “support.h”
#include “snmpv3.h”
. . .

/* This structure is defined as external in SNMPAgentTask() */
const AGENT_CONTEXT snmp_ac =
{
    mibs, (sizeof(mibs) / sizeof(MIB *)),
    thosts, (sizeof(thosts) / sizeof(TRAP_HOST)), 0, COLDSTART,
    &TM_DPI
};

void main(void)
{
    Ninit();
    Portinit(“*”);
    ussSNMPAgentTask();
    Nterm();
}
```

ussSNMPAgentInit

Initializes the agent.

```
sint16 ussSNMPAgentInit(const AGENT_CONTEXT *acp);
```

This function initializes the agent with the run-time environment defined by the value of the `AGENT_CONTEXT` parameter. The run-time environment that the agent uses is defined by the MIBs visible to the agent, the Trap hosts, and a transport mapping.

Return Value

>= 0	No error
< 0	An error

Example

```
#include “snmpv3.h”
. . .
extern const AGENT_CONTEXT snmp_ac;
. . .
i1 = ussSNMPAgentInit(&snmp_ac);
if (i1 < 0)
{
#if NTRACE
   Nprintf(“SNMPAgentTask: Initialization failed %d\n”, i1);
#endif
   return;
}
```

ussSNMPAgentCheck

Checks the status of the agent for pending requests, and responds as necessary.

```
sint16 ussSNMPAgentCheck(void);
```

This function checks the transport for incoming messages, and generates responses as necessary.

Return Value

>= 0	No error
< 0	An error

Example

```
#include “snmpv3.h”
. . .
/* Control loop for reading requests and
   forming/sending replies */
while (ussSNMPAgentCheck() >= 0)
    ;
```

ussSNMPAgentShut

Terminates the agent.

```
sint16 ussSNMPAgentShut(void);
```

This function performs any clean-up necessary to terminate all the layers of the Agent.

Return Value

>= 0	No error
< 0	An error

Example

```
#include “snmpv3.h”
. . .
ussSNMPAgentShut();
```

ussSNMPAgentTrap

Sends a trap to all configured trap hosts as defined in the AGENT_CONTEXT.

```
sint16 ussSNMPAgentTrap(uint8 type, uint8 spec,
                        const uint8 *contextName,
                        const uint8 *vbs, uint16 len);
```

type	the trap type
spec	trap-specific code
contextName	context or community name
vbs	pointer to a variable bindings for trap
len	the buffer length

The ***ussSNMPAgentTrap()*** function may be used from an agent application to send a trap to a manager. Trap types specified as 0 through 6 are shown in Table 2.

Table 2: SNMP Trap Types

Value	Trap Type	Description
0.	cold start	The agent network protocol has reinitialized, indicating that its configuration may have been altered.
1.	warm start	The agent network protocol has reinitialized; however, its configuration has not been altered.
2.	link down	A communication link has failed. The failing link is identified via the first variable within the variable bindings field of the PDU (protocol data unit). The PDU is, essentially, the data protocol used by SNMP. The variable bindings field is a list of MIB variables sent to the manager packaged within a PDU.
3.	link up	A communication link has come up. The affected link is identified as the first element within the variable bindings field.
4.	`AuthenticationFailure`	The agent could not resolve the authentication for an SNMP message received from the manager.
5.	`EgpNeighborLoss`	An EGP peer neighbor is down.
6.	`EnterpriseSpecific`	A nongeneric trap has occurred. This is specific to a particular enterprise. Use this for application-specific traps.

Return Value

The number of traps sent. This should be compared to the number of trap hosts configured in the AGENT_CONTEXT.

Example

To send a trap from an application, simply call ***ussSNMPAgentTrap()*** and pass in the trap type, the trap-specific code, the context/community name, a pointer to a buffer of variable data for the manager to process, and the length of the variable data. If the buffer is not needed 0 may be used. For example, to send a "warm start" trap with no variable data, use:

```
int rc;           /*return code */
rc = ussSNMPAgentTrap(WARMSTART,0, "public", 0, 0);
if (rc <= 0)
    <process error >
```

If a trap must pass variable data to the manager, declare a buffer, assign the variable binding data to it and pass it to ***ussSNMPAgentTrap()***.

```
#define VARBUFFERSIZE   <some constant value>
....
int rc;           /* return code */
unsigned char varbuffer[VARBUFFERSIZE];
....
varbuffer = <load the data into the buffer>;
....
rc = ussSNMPAgentTrap(WARM_START, 0, "public", varbuffer,
                VARBUFFERSIZE);
if (rc != 0)
   <process error>;
```

This function call is flexible in that the variable data may be passed in any format; however, it is constrained to what the manager can understand. Generally, this would be in the form of an SNMP variable bind list.

Customizing the Agent

Configuring the Agent MIB

Standard MIBs are supplied with SNMP/USNET based on Internet standards defined by RFCs (request for comments, on the Internet) 1156 and 1213. These RFCs have since been clarified in several updated RFCs modularized from the originals.

MIB Structure

Each MIB module must be molded into the MIB structure used by the agent.

```
typedef struct
{
    const MIBVAR *mvp;          /* MIB variables */
    sint16 (*numvars)(void);    /* Number of variables */
    const MIBTAB *mtp;          /* MIB tables */
    sint16 (*numtabs)(void);    /* Number of tables */
    void (*get)(sint16 varix, sint16 tabix, uint8 **vvptr);
    sint16 (*set)(sint16 varix, sint16 tabix);
    sint16 (*index)(sint16 varix, sint16 index);
    void (*init)(uint16 type);  /* Initialize the MIB */
} MIB;
```

MIBVAR and MIBTAB Structures

The MIBVAR and MIBTAB structures are the primary data structures, which define MIB data. Each MIB contains variables *mibvar* and *mibtab,* which are simply arrays of these structures. MIBVAR and MIBTAB are defined in **snmpv3.h** as follows:

```
typedef struct
{
    uint8 nlen, name[MAXOID];
} OID;

typedef struct
{
    OID oid;              /* Base OID of table */
    uint8 nix;            /* Number of indices for table */
    uint16 ix[MAXKEY];    /* Index values (offsets) */
    uint16 len;           /* Length of table */
} MIBTAB;

typedef struct
{
    OID oid;              /* Identifier name, length */
    uint8 opt;            /* Options */
    uint8 type;           /* Type of variable */
    sint16 len;           /* Length of pointer field */
    void *ptr;            /* Pointer to variable data */
} MIBVAR;
```

MIBVAR contains the definitions and values of all MIB variables. MIBTAB contains indices into the MIBVAR for accessing MIB table (SEQUENCE OF) entries. Most of these fields are used internally by the SNMP agent; however, some are useful to know. OID is used to uniquely define each record in the MIBVAR and MIBTAB. Also, for a given MIB table variable, the OID is the key value, which links MIBVAR and MIBTAB entries. The purpose of the MIBVAR is simply to store all MIB data; that is, scalar values and values within a MIB table. In the case of a MIB table, the mibtab.ix[i] values are used as indices to the appropriate records in the MIBVAR. An example of its use is provided in the 'MIB.index()' section.

Default Operation

When the SNMP agent receives a ***GetRequest*** PDU (protocol data unit), the entries in the `MIBVAR` array are reviewed to find an entry that matches the requested OID. The *`ptr`* field in the matching entry is then used to locate the memory location that contains the value that should be returned. For scalar variables, this location is read directly. For variables in tables, an offset is added to the pointer that corresponds to the index portion of the OID in the ***GetRequest*** PDU.

When the SNMP agent receives a ***SetRequest*** PDU, the corresponding entry is located as above, and the memory location based on the *`ptr`* field is overwritten with the value provided in the ***SetRequest*** PDU.

MIBVAR Record Options

Some of the variables in `MIBVAR` may not be well suited to the default operation of the SNMP agent. To support these needs, the *`opt`* field of the `MIBVAR` record allows for flags that will indicate that special processing is required.

`IMMED` The variable value is stored directly in the *`len`* field, rather than being pointed to by the *`ptr`* field. The variable should be an 8-bit value. The value for *`ptr`* can be 0.

`IMMED2` The variable value is stored directly in the *`type`* and *`len`* fields, rather than being pointed to by the *`ptr`* field. The variable should be a 16-bit value. The value for *`ptr`* can be 0.

`SCALAR` The variable is in a table, but should be looked up without adding an index to *`ptr`*. This allows a variable to be part of a table, but not accessed in the same manner as other variables in the table. If the value for a variable is known to be the same for every index in the table, then this technique can be used to reduce the size of the memory image that represents the contents of the table. This flag need not be specified for normal scalar variables.

`W` The variable may be modified.

`SX` The variable is the first item of a MIB table.

`CAR` A read notification function may be called before returning the value of the variable.

`CAW` A write notification function may be called after writing a new value to the variable.

`CHOICE` A '`CHOICE`' ASN.1 syntax element is required in the OID of this object. Note that it is only used to force the `atTable` to behave correctly and, if defined, code size will increase for all MIBs.

MIB.set() and MIB.get() Functions

These functions are written as part of each MIB and provide the actions to perform for read or write notification.

```
static sint16 set(sint16 varix, sint16 tabix);
void get(sint16 varix, sint16 tabix, uint8 **vvptr);
```

The first argument, *varix*, is an integer which acts as an index into the MIB identifying the variable to be accessed. If that MIB variable is a MIB table, the *tabix* parameter may be used as a 0-based index into the table. If *varix* is a scalar value or not a table entry, then no index is required and -1 is passed in for *tabix*. The ***vvptr* is passed to the ***get()*** function in case the MIB needs to replace the value pointer with a new address for the agent to operate upon.

The value returned by ***set()*** should be 0 if the function executes normally. In the case of an error situation, the value returned from these functions will be used as an error code in the response that the SNMP agent sends to the SNMP request.

The ***get()*** and ***set()*** functions are called indirectly from the function ***ussSNMPAgentCheck()*** in **agent.c** through the MIB structure in which the ***get()*** function pointer resides. The declaration below shows how the MIB structure is defined.

Example

```
#include “snmpv3.h”
. . .
static void get(sint16 varix, sint16 tabix, uint8 **vvptr)
{
    const MIBVAR *mvp = &mibvar[varix];
    uint8 *bytevp = *vvptr;

    /*
    ** If varix is 3, the variable is a 32-bit value
    ** that must be updated before being read by the agent.
    ** We set it here to a value that is determined by using
    ** a value in a table indexed by an array of index
    ** values.
    */

    if (varix == 3)    /* Fourth variable in MIB */
    {
        *(uint32 *)*vvptr = Barray[Aarray[tabix].nindex].value32;
    }

    /*
    ** If varix is 12, the first index is not stored in the
    ** table. The second and all subsequent indices are in
    ** the table, however.  We can simply point the value
    ** pointer to a new location.
    */
```

```
   if (varix == 12)   /* Thirteenth variable in MIB */
    {
        if (tabix == 0)
            *vvptr = &value;
        else
            *vvptr = &table[tabix].value;
    }
}

static sint16 set(sint16 varix, sint16 tabix)
{
    MIBVAR *mvp = &mibvar[varix];
    uint8 *bytevp = mvp->ptr;

    if (varix == 3)
    {
        if (*(uint32 *)bytevp == 0x1234567)
        {
            *(uint32 *)bytevp = 0;
            return badValue;
        }
    }

    return 0;
}
. . .

const MIB mib_example =
{
    mibvar,
    mibvarsize,
    mibtab,
    mibtabsize,
    get,
    set,
    index,
    init
};
```

The globally-accessible function pointer `mib_example.get` is assigned the ***get()*** function which is local to the current MIB module. The ***mib_example.get()*** function is only called if `CAR` is in the option field for the variable and the ***get()*** function pointer is valid (that is, not 0). Upon entry into the ***get()*** function, the variable *`varix`* is an index into the `MIBVAR` array for the current variable to be read. The *`tabix`* is assigned –1 if no table is being accessed. Otherwise, *`tabix`* is a zero-based index into the table to which the variable belongs.

MIB.index() Function

Determines size of tables in a MIB.

```
sint16 index(sint16 varix, sint16 index);
```

If tables exist in a MIB, the SNMP agent needs a mechanism to determine the size of the tables that have been added. The ***index()*** function indicates when the end of the table has been reached and also can be used to specify when a table entry should be skipped. Good examples of ***MIB index()*** functions can be found in **mib_if.index, mib_tcp.index, mib_udp.index**, etc.

The ***index()*** function is required to implement a table.

When the SNMP agent receives a get request or a get-next request that involves a MIB table and the ***index()*** function is defined, the agent will call the ***index()*** function while iterating through the table to determine if an entry should be included in the search for the variable. The MIB ***index()*** function is defined similarly to the MIB ***get()*** and ***set()*** functions.

Return Value:

1	Accept the record
0	Skip over the record
-1	End of table

Example

```
/* Index the IP MIB's tables */
static sint16 mibindex_ip(sint16 varix, sint16 tabix)
{
    uint8 *cp;
    uint16 us1;
    sint16 i1;

    cp = (uint8 *)mibvar_ip[varix].oid.name + 5;
    us1 = *cp++ << 8;
    us1 += *cp;
```

```
    switch (us1)
    {
    case 0x0416:            /* IP net to media table */
        if (tabix >= NCONFIGS)
            goto lab7;
        if (netconf[tabix].ncstat == 0)
            break;
        for (i1 = 0; i1 < Eid_SZ; i1++)
            if (netconf[tabix].Eaddr.c[i1])
                goto lab5;
        break;
    case 0x0414:            /* IP address table */
        if (tabix >= confsiz)
            goto lab7;
        if (netconf[tabix].flags & LOCALHOST)
            goto lab5;
        break;
    case 0x0415:            /* IP routing table */
        if (tabix >= NCONFIGS)
            goto lab7;
        if (netconf[tabix].ncstat == 0)
            break;
        if (!(netconf[tabix].flags & LOCALHOST))
            goto lab5;
        break;
    default:                    /* any other */
        goto lab5;
    }
    return 0;
lab5:
    return 1;
lab7:
    return -1;
}
```

In this example, a section of the Object ID is used to identify the variable for which the index function is being called. The value of *index* could also be used for this purpose, but using a section of the OID allows a subtree of the MIB to easily be identified. At the beginning of the function, *cp* is set up to point to the interesting section of the OID, and then the next two bytes of the OID are stored in *us1*.

This is just one example of how an ***index()*** routine could be coded. Processing of `accept`, `skip`, or `end of table` is determined by checking values of USNET data structures in the above case. The *`index`* may be used as an index into some of these structures. The `MIBTAB` values are simply used as flags to indicate which variable is to be processed. The actual value of the variable requires accessing of the USNET data structures. Refer to the USNET documentation and source code for explanations of values such as *`NCONFIGS`*, and `netconf[`*`tabix`*`]`.

Adding New MIBs

A particular application may require new MIBs in addition to those supplied as part of the MIB-II. If this is the case, use the ASN.1 (Abstract Syntax Notation) syntax to add the definitions of variables to a MIB file. Refer to a text on SNMP or the appropriate RFCs for definitions of this syntax. Then use **MIBTOC** to translate the ASN.1 definitions into C code understandable to the SNMP agent.

MIB Translation Overview

To use a new MIB with the SNMP/USNET agent, a file describing the MIB variables must be compiled into C source code. The program **MIBTOC**, performs this translation. It reads a description of the MIB variables in ASN.1 format, and produces two ANSI C-compatible files. In the following diagram, “MIB” represents the name of the MIB file.

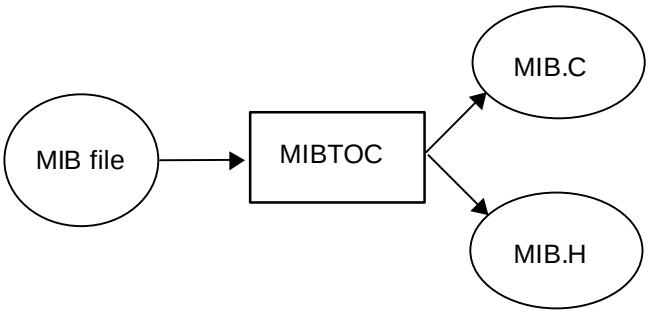

Figure 2: MIB Translation

The application can compile and link the MIB with the agent so the agent can access the MIB database.

Building the MIB Translator

The translator is provided as source code, which ought to be compiled before use. It is located in the **snmpsrc\\tool**s directory. To build it by hand, simply pass the source file as an argument to a compiler/linker. For instance, if using the Borland compiler, run:

```
bcc snmpsrc\tools\mibtoc.c
```

Or, if building from a UNIX environment, run:

```
cc snmpsrc/tools/mibtoc.c
```

MIBTOC is ANSI-compatible and can be compiled by most commercially available compilers. Since the **MIBTOC** application uses a significant amount of stack space, the compiler or linker may need to be configured with an option to increase the stack space. The compiler is included in executable format for DOS and Windows platforms.

Running the MIB Translator

MIBTOC takes one or two arguments: The first argument is the name of the MIB file to be processed, and the optional second argument provides the base name for the output file. The syntax is:

```
MIBTOC mibfile [outfile]
```

If an output file name is not specified, the name for the output files will be derived from the base file name of the input file. For example, this command will generate the output files **toaster.c** and **toaster.h**:

```
MIBTOC toaster.mib
```

If the second parameter is provided, then the output file names are based on the second parameter. Given this command line, the translator will generate the output files **test.c** and **test.h**:

```
MIBTOC toaster.mib test
```

Watch the output of **MIBTOC** to be sure that no errors occurred in preparing the output files. A normal run will look like:

```
C:\usnet\snmpsrc>mibtoc rfc2571.txt

USNET MIB to C Translator 1.10
  Copyright (c) U S Software 1994, 1999, 2000.
Root: ccitt
Root: iso
Root: joint-iso-ccitt
Type 'No Access': org { iso 3 }
Type 'No Access': dod { org 6 }
Type 'No Access': internet { dod 1 }
Type 'No Access': mgmt { internet 2 }
Type 'No Access': experimental { internet 3 }
Type 'No Access': private { internet 4 }
Type 'No Access': security { internet 5 }
Type 'No Access': snmpV2 { internet 6 }
Type 'No Access': snmpDomains { snmpV2 1 }
Type 'No Access': snmpProxys { snmpV2 2 }
Type 'No Access': snmpModules { snmpV2 3 }
Type 'No Access': mib-2 { mgmt 1 }
Type 'No Access': transmission { mib-2 10 }
```

```
Type ‘No Access’: enterprises { private 1 }
Type ‘No Access’: snmpFrameworkMIB { snmpModules 10 }
TC: SnmpEngineID (OctetString)
TC: SnmpSecurityModel (Integer)
TC: SnmpMessageProcessingModel (Integer)
TC: SnmpSecurityLevel (Integer)
TC: SnmpAdminString (OctetString)
Type ‘No Access’: snmpFrameworkAdmin { snmpFrameworkMIB 1 }
Type ‘No Access’: snmpFrameworkMIBObjects { snmpFrameworkMIB 2 }
Type ‘No Access’: snmpFrameworkMIBConformance { snmpFrameworkMIB 3 }
Type ‘No Access’: snmpEngine { snmpFrameworkMIBObjects 1 }
Type ‘OctetString’: snmpEngineID { snmpEngine 1 }
Type ‘Integer’: snmpEngineBoots { snmpEngine 2 }
Type ‘Integer’: snmpEngineTime { snmpEngine 3 }
Type ‘Integer’: snmpEngineMaxMessageSize { snmpEngine 4 }
Type ‘No Access’: snmpAuthProtocols { snmpFrameworkAdmin 1 }
Type ‘No Access’: snmpPrivProtocols { snmpFrameworkAdmin 2 }
Type ‘No Access’: snmpFrameworkMIBCompliances {
snmpFrameworkMIBConformance 1 }
Type ‘No Access’: snmpFrameworkMIBGroups { snmpFrameworkMIBConformance 2
}
Type ‘No Access’: snmpEngineGroup { snmpFrameworkMIBGroups 1 }
2554 lines processed OK
```

If there is a problem in processing the file, the last line will not read "`. . . processed OK`" but rather will describe an error in processing the file. For example, if the definition for ***MAXOID*** in **mibtoc.c** is too small, then this message will be displayed:

```
L388 myTableIndex MAXOID too small
```

This indicates that in processing line 388 of the MIB file, it was discovered that there was not enough room to build the needed Object ID array. To correct this, the value for ***MAXOID*** should be increased in **mibtoc.c**, and **MIBTOC** should be rebuilt. Also ***MAXOID*** should be increased to the same value in **snmpconf.h**, because it will be used again when building the SNMP agent.

MIB Files

MIBTOC generates two files as output. Using the example of an ASN.1 input file named **toaster.mib**, the output files would be **toaster.c** and **toaster.h**. The SNMP agent uses the output files as follows:

toaster.h Defines external variable and symbol definitions to which the application and MIB module may wish to refer as "`extern`".

toaster.c Allocates MIB variable and table values statically and provides the global '`MIB mib_toaster`' structure declaration to provide global access to the MIB from the application.

Read/Write Notification

Each variable in a MIB may have read or write notification associated with it. This means that prior to a get operation or after a set operation, the agent will signal the MIB that its data is being operated upon.

For ***get-***, ***getNext-*** or ***getBulk-***requests, the option field in the MIB variable is checked for read notification (`CAR` – Call Application Read). If this is set for the variable, the ***get()*** function for the MIB will be called with the index of the variable and a pointer to a pointer to the value of the variable. This is so that the MIB can update the value of the variable or dynamically redirect it to a new memory location.

For ***set-***requests, the option field in the MIB variable is checked for write notification (`CAW` – Call Application Write). If this is set for the variable, the MIB ***set()*** function will be called with the index of the variable. Special processing can be performed due to important changes in the value of the MIB variable.

To indicate to the agent that read or write notification is required on a given variable, add the `CAR` and/or `CAW` options to the `opt` field of the variable record within the MIB source file using the bitwise `OR` operator (i.e. '|').

Example

```
{8,{0x2b,6,1,2,1,1,6,0}, W | CAR | CAW, String,
        sizeof(syslocat), syslocat}, /* sysLocation */
```

This example shows a `MIBVAR` record (see the next section) which adds read and write notification to the MIB variable *`sysLocation`*. Before modification, the option field was simply `W`, indicating a variable that allows write access. The option field may be zero for no options or a combination of others. The possibilities are defined in **snmpv3.h** and are shown in Table 3 below.

```
#define IMMED   0x01 /* Immediate value in mvp->len */
#define IMMED2  0x02 /* Immediate value in mvp->type + len */
#define BASE1   0x03 /* Base 0 in data space, base 1 in MIB */
#define SCALAR  0x04 /* Table not indexed (no offset) */
#define W       0x80 /* Write allowed */
#define SX      0x40 /* Sequential table index inferred */
#define NWORDER 0x20 /* Network byte ordering for basic type */
#define CAR     0x10 /* Call application after read */
#define CAW     0x08 /* Call application before write */
```

Table 3: MIBVAR Record Options Field

Options Field	**Description**
`IMMED`	The variable value is stored directly in the *`len`* field (see below), rather than using the *`ptr`* field to store the address of the value.
`IMMED2`	Similar to `IMMED` except the variable value is stored directly in the type and *`len`* fields (see below).
`BASE1`	The variable index value is represented by SNMP starting at a base value of '1' even though the agent must deal with the actual data with a base '0'.
`SCALAR`	A scalar value. In other words, the value is not in a table even though its ASN.1 definition defines it as part of a table.
`W`	A variable that allows write access, i.e., the value may be modified.
`SX`	Indicates the first item of a MIB table, i.e., a `SEQUENCE OF`.
`CAR`	Use Read notification.
`CAW`	Use Write notification.

Summary of Adding a User-Defined MIB

1. Create the standard "out of the box" version of the SNMP agent, and confirm that the standard MIB-II variables are accessible from an SNMP manager.
2. Build the **MIBTOC** compiler, if it is not already built for the development platform.
3. Create the enterprise-specific MIB. This example presents the wt2000 remotely accessible weather station MIB, which uses the MIB called **weather.mib**. The MIB will be associated with a product of the fictional company "WeatherTek International" that makes devices that record weather conditions. These conditions can be retrieved from their instruments through an SNMP manager.

The first information to be included in the user-defined MIB will establish the path in the MIB hierarchy to the enterprise-specific MIB. If the enterprise code for WeatherTek International were 123, and the variables were those collected by the wt2000 model, then the following information might appear first in **weather.mib**:

```
—                   MIB DESCRIPTION
WEATHER-MIB DEFINITIONS ::= BEGIN
—
weathertek      OBJECT IDENTIFIER  ::= { enterprises 123 }
wt2000          OBJECT IDENTIFIER  ::= { weathertek 3 }
```

In this example, the weather station contains components that monitor conditions at a number of altitudes. Some of the variables in **weather.mib** concern the weather station as a whole, and some concern the conditions at each altitude. Let us say that a string is set up to hold the unit location, and the latitude and longitude of the installation are also stored.

This information might appear in **weather.mib** as follows:

```
—
— The wt2000 Group
—
location OBJECT-TYPE
   SYNTAX      DisplayString
   ACCESS      read-write
   STATUS      mandatory
   DESCRIPTION “The geographical name for the device location.”
   ::= { wt2000 1 }
latitude OBJECT-TYPE
   SYNTAX      INTEGER
   ACCESS      read-write
   STATUS      mandatory
   DESCRIPTION “The latitude at which the device is installed.”
   ::= { wt2000 2 }
longitude OBJECT-TYPE
   SYNTAX      INTEGER
   ACCESS      read-write
   STATUS      mandatory
   DESCRIPTION “The longitude at which the device is installed.”
   ::= { wt2000 3 }
```

Now a table can be introduced to hold the information that is collected for a number of altitudes. For this table, the altitude will act as an index, and temperature, humidity, wind speed and wind direction will be monitored. Here is how it might appear in **weather.mib**:

```
weatherTable OBJECT-TYPE
   SYNTAX      SEQUENCE OF weatherEntry
   ACCESS      not-accessible
   STATUS      mandatory
   DESCRIPTION “This table contains a tally of weather conditions”
   ::= { wt2000 4 }
weatherEntry OBJECT-TYPE
   SYNTAX      WeatherEntry
   ACCESS      not-accessible
   STATUS      mandatory
   DESCRIPTION “Each row represents conditions at a given altitude.”
   INDEX       { altitude }
   ::= { weatherTable 1 }
WeatherEntry ::= SEQUENCE {
   altitude      INTEGER,
   temperature   INTEGER,
   humidity      INTEGER,
   windSpeed     INTEGER,
   windDirection INTEGER { NORTH     (1),
                           NORTHEAST (2),
                           EAST      (3),
                           SOUTHEAST (4),
                           SOUTH     (5),
                           SOUTHWEST (6),
                           WEST      (7),
                           NORTHWEST (8)}}
altitude OBJECT-TYPE
   SYNTAX      INTEGER
   ACCESS      read-only
   STATUS      mandatory
   DESCRIPTION “Altitude in meters, used as an index.”
   ::= { weatherEntry 1 }
```

The definitions for *temperature*, *humidity*, *windSpeed*, and *windDirection* would appear similar to the definition for *altitude*.

Process the MIB with **MIBTOC** to create source code. Make sure that the compiler reports no errors. Using the **mibtoc.exe** file in the **snmpsrc\tools\windows** directory:

```
cd snmpsrc
tools\windows\mibtoc weather.mib
```

Edit **snmpsrc\makefile** to specify the files generated by **MIBTOC**. So in this example, add the line '`FILES += weather`' in the midst of the other.

```
cd snmpsrc
edit makefile
     FILES += mib_usn
     FILES += weather # <<< New >>>
     FILES += snmp
make
```

If there are any tables in the user-defined MIB, an ***index()*** function will have to be created in **snmpsrc\weather.c** and added to the MIB **mib_weather** declaration.

```
cd snmpsrc
edit weather.c
     const MIB mib_weather =
     {
         mibvar,
         mibvarsize,
         mibtab,
         mibtabsize,
         0,          /* get */
         0,          /* set */
         index,      /* <<< New >>> */
         0           /* init */
     };
```

Declare the program variables that are introduced in the user defined MIB. In this example, external declarations for the variables will be written into weather.h, but the variables will not be declared in any module. The names of the variables are based on the names appearing in the MIB definition, and can be found in weather.h, which is excerpted here:

```
extern char *location;
extern int latitude;
extern int longitude;
extern struct weatherTable weatherTable[];
```

These variables must be declared somewhere in the application, and for this example the declarations are made in a modified version of **weather.c**:

```
#define WTABSZ 3  /* number of entries in weather table */
char *location;
int latitude;
int longitude;
struct weatherTable weatherTable[WTABSZ];
```

Note that the size of the table is not apparent from the information in the MIB definition and may be variable. In this example, a constant has been defined to specify the size. *WTABSZ* represents the largest possible table size. This information should be used by the ***index()*** function.

Initialize the variables in the user-defined MIB. Any default values or fixed values can be set up before the SNMP agent is started. Also, any index fields in tables must be initialized before the agent is started.

Here is an example from the modified **weather.c**:

```
const char defaultlocation[] = “Portland, Oregon”;
#define DEFAULTLATITUDE 46
#define DEFAULTLONGITUDE 123

static void init(uint16 type)
{
    memset(weatherTable, 0, sizeof(weatherTable));
    location = defaultlocation;
    latitude = DEFAULTLATITUDE;
    longitude = DEFAULTLONGITUDE;

    for (i1 = 0; i1 < WTABSZ; i1++) {
        weatherTable[i1].altitude = i1 * 1000 + 1000;
        weatherTable[i1].windDirection = 1;
    }

}
```

In this example, default values for *location*, *latitude*, *longitude* and the *windDirection* field in *weatherTable* are initialized. The *altitude* index field in the table is initialized with the values 1000, 2000 and 3000.

If the value of a variable should be updated before being read, then the ***get()*** function should be implemented.

Likewise, if special action should be taken once a variable is written, then the ***set()*** function should be implemented, and if the number of rows in a table is variable then the ***index()*** function should be implemented.

The `weather` MIB structure will have to be updated to reflect any required get, set, index or init functions:

```
const MIB mib_weather =
{
    mibvar,
    mibvarsize,
    mibtab,
    mibtabsize,
    get,       /* <<< New >>> */
    set,       /* <<< New >>> */
    index,     /* <<< New >>> */
    init /* <<< New >>> */
};
```

Configuring the Transport Mapping

A Transport Mapping is a defined method of data transfer between SNMP hosts. RFC 1906 defines the use of SNMP over UDP/IP on Internet-based networks as well as many others. From this, a module was defined called `TRANSPORT_MAPPING`. Here is the structure definition that the SNMP/USNET agent uses:

```
typedef struct
{
   /* Initialize underlying transport framework */
    sint16 (*init)(uint8 *ip, uint32 *maxsize, uint8 *name);

   /* Open passively to receive SNMP messages */
    sint16 (*passive_open)(void);
    sint16 (*passive_read)(uint8 *buff, uint16 len);
    sint16 (*passive_write)(const uint8 *buff, uint16 len);
    sint16 (*passive_close)(void);

   /* Open actively to send SNMP messages */
    sint16 (*active_open)(const uint8 *rhost);
    sint16 (*active_write)(const uint8 *buff, uint16 len);
    sint16 (*active_read)(uint8 *buff, uint16 len);
    sint16 (*active_close)(void);

   /* The host's system time */
    uint32 (*time)(void);
} TRANSPORT_MAPPING;
```

The application is expected to perform basic initialization of the network or other media. Once that is completed, the agent may perform the following operations:

init() Initialize the transport specific features required by the agent. Included are the IP address, maximum message size, and host name. If any of these is defined and does not conflict with the transport layer, they can remain the same.

passive_open() Tell the transport that the agent is ready to receive data.

passive_read()	Get available data from the transport.
passive_write()	Transmit potential responses to ***passive_read()*** operations.
passive_close()	Tell the transport that the agent will no longer receive data.
active_open()	Tell the transport to create a data channel to a particular host for sending traps. Note that the `rhost` field is one of the trap hosts defined by the application.
active_write()	Transmit a message to the host to which an ***active_open()*** was performed.
active_read()	Receive data on the trap channel. This will not occur with SNMPv1 and v2c. However, SNMPv3 has the provision that an agent may have to authenticate itself to a management station. Version 3 trap packets are not supported at this time.
active_close()	Close the data channel for writing traps.
time()	Get the system time in tenths of a second.

Each of the above operations returns a signed 16-bit value, except ***time()*** which returns the current time as a 32-bit value. For ***passive_open(), passive_close(), active_open()***, and ***active_close()*** the return value should be >= 0 unless an error occurs. For ***passive_read(), passive_write(), active_read()***, and ***active_write()*** functions the return value should represent the number of bytes transmitted or received. Note that the agent cannot internally handle an error value when performing ***passive_open()***. Essentially, the agent is useless without its passive functions.

When the ***ussSNMPAgentTrap()*** function is called by the application or by the agent, the agent will actually iterate through each ***active_xxx()*** function for each trap host.

For example implementations, see the following:

Snmpsrc\tm_bsd.c	USNET BSD socket interface (USNET, UNIX, and Windows)
Snmpsrc\tm_dpi.c	USNET DPI interface

Index