DMH software

P.O Box 2714
Acton, MA 01720

Voice: 978-263-0526
Fax: 206-202-3741
info@dmhsoftware.com
http://www.dmhsoftware.com

DMH SNMP

The Portable Advanced SNMP Agent SDK

Product Description

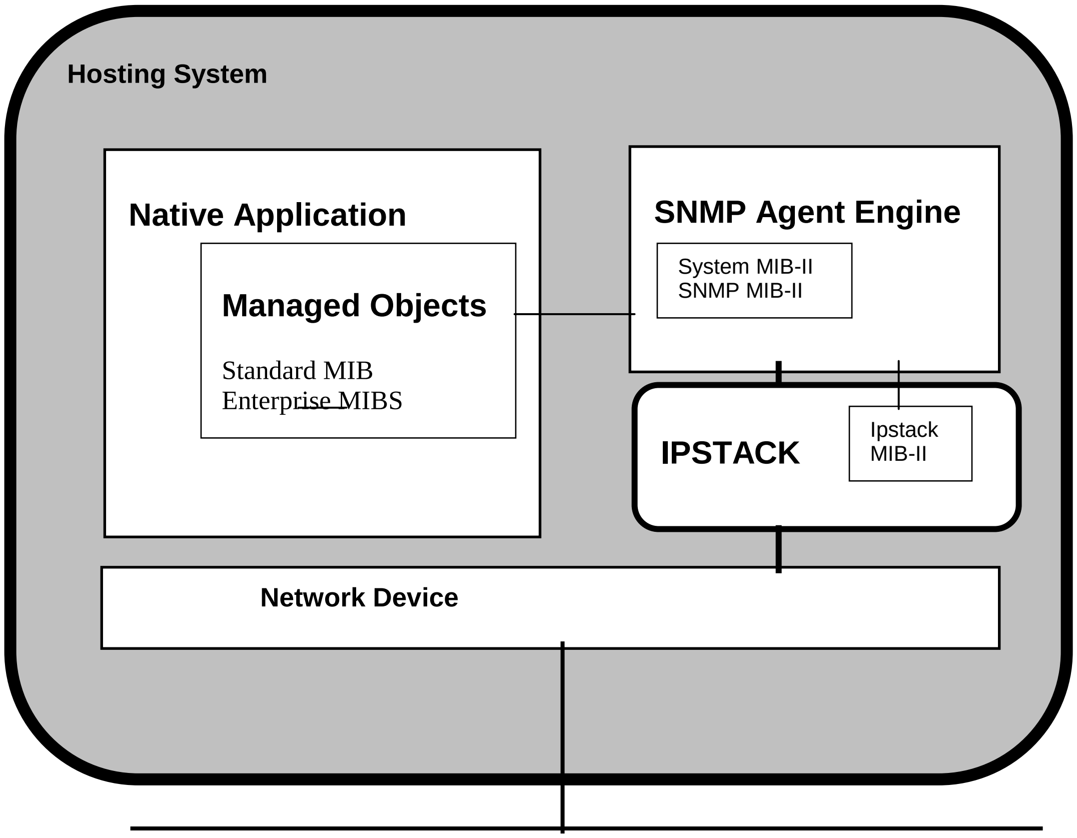
Document Version: 1.7
March 2007

The SNMP Protocol

The Simple Network Management Protocol (SNMP) is today a de-facto industry standard for monitoring and managing 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 centrally monitor, diagnose and configure each such device across local and wide area networks. SNMP is the protocol that enables this interaction.

In order for a device to respond to SNMP requests, it must be equipped with the software that enables it to properly interpret an SNMP request, perform the actions required by that request, and produce an SNMP reply. The SNMP Agent is a subsystem software module residing in a network-entity.

The following figure shows a typical system composed of a native application and its managed objects. We can see how SNMP interacts with those objects, making them accessible to a central SNMP manager through the network.



The collection of related objects implemented by a system is called a MIB: Management Information Base. A MIB is defined as a specification in a formal language called ASN.1. MIB-II is the most basic and most popular MIB; all SNMP agents implement parts of MIB II. Examples of other MIB's are:

- *Ethernet MIB*, which focuses on Ethernet interfaces
- *Bridge MIB*, which defines objects for the management of 802.1D bridges
- *RMON MIB*, which allows sophisticated line monitoring activities to be performed remotely and reported to a central application via SNMP

DMH Advanced SNMP Agent Highlights

As developers of SNMP applications for the last 10 years, we found that the most critical aspects of an SNMP system are:

1. The large amount of work involved in implementing MIB modules.
2. The heavy CPU overhead involved in the processing of SNMP commands.

As a result, we have designed a highly portable, modular, and extensible SNMP system. Our product aims to maximize programmer productivity, minimizing the SNMP knowledge required and improving run time efficiency.

DMH SNMP Agent Advantages:

- ✓ **Highly Deployed:** Used in many commercial products and integrated in different types of systems.
- ✓ **Well maintained and well supported:** Customers receive excellent prompt support and all required information. Reported problem are fixed immediately.
- ✓ **One of the best in the industry:** Meets high quality standards. Code and documentation are clear and well `maintained`. Excellent run-time performance.
- ✓ **Resilient:** Stands tall against popular industry snmp testing systems. Meets DOCSIS snmpv3 requirements, including snmpv3 application MIBS.
- ✓ **Fast Integration:** Porting the basic kernel is done in a few days. Open and flexible architecture. Designed to be integrated with any given system including real-time embedded Systems.
- ✓ **MIB Compiler:** The SDK includes SMIv2 MIB Compiler, which produces 'C' code from standard ASN.1 MIB files. The Compiler generates most of the code needed for the support of mib objects. Mib development is very easy and fast.
- ✓ **Ideal for new MIB development**: The agent is designed to ease the process of adding the support for your own mib objects (scalars and tabular mibs). Convenient facilities for instance evaluation and table manipulation.
- ✓ **Highly efficient implementation:** a MIB search involves an average of 30 compare operations (in contrast with over 1000 in popular SNMP implementations).
- ✓ **Dynamic object registration**: a "must" for systems requiring high modularity and versatility.
- ✓ **SNMPv2c** protocol fully supported (get-bulk, error-codes, 64bit counters, rowStatus).
- ✓ **SNMPv3** protocol fully supported: USM, MD5, SHA1, DES, AES, VACM, and SNMPv3 Application MIBs
- ✓ **MIB-II** System Group and SNMP Group are built-in.
- ✓ **Portable IpStack MIB-II:** Generic implementation designed for any given ipstack.
- ✓ **Develop and Debug**: Can be compiled and work on your software development platform (e.g. w32, Linux, Solaris etc). Mib development and debugging can be done on the development system. Once developed, you can compile for the target system
- ✓ **CPU Range:** Can be compiled and integrated in many kind of CPUs architectures including: 64, 32, 16 and 8 bits CPUs. The agent is used in small 16/8 bit CPU systems, as well as on 64 bit CPU systems.
- ✓ **Services to customers: Integration**, MIB design and development, MIB conversion (from other snmp agents), Changes to meet special requirements.

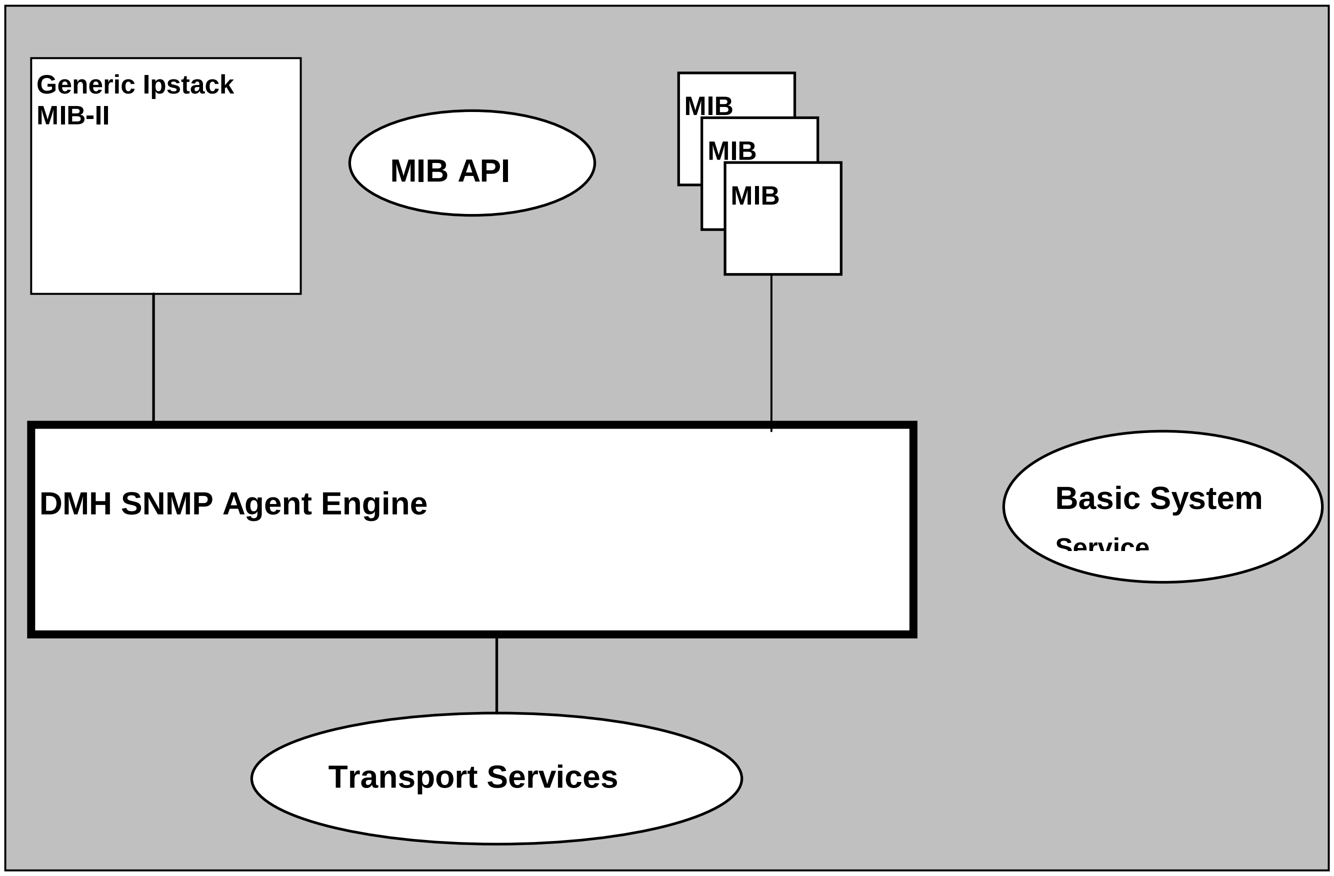
DMH SNMP Agent Architecture

The DMH SNMP portable Agent consists of the software components implementing the SNMP protocol and MIB II. This is the Agent kernel.

The kernel is relatively small, simple, and highly portable. Its overall design is optimized for fast integration and ease of use.

The SDK includes the DMH MIB-Compiler for rapid development of additional MIBs. The MIB compiler accepts an ASN.1 MIB definition as input, and produces a set of 'C' and 'H' files as output. The generated 'C" files include the code implementing the given MIB.

The following diagram illustrates the DMH SNMP internal architecture:



▣ **DMH SNMP Module w/out user code**

□ **DMH SNMP Module with placeholders for user code.**

◯ **System modules interacting with DMH SNMP Agent**

Main Components

1. The DMH SNMP kernel. This is the heart of DMH SNMP Agent. It processes incoming requests, identifies the various parts of a request, checks for proper command syntax, and eventually interacts with the MIB modules to retrieve or modify the value of a requested object. The kernel interacts with the underlying transport subsystem (usually UDP/IP) for reception and transmission of SNMP datagrams.

2. MIB-II: The System and SNMP Groups defined by MIB-II are integral part of the SNMP Agent. The implementation is based on the MIB-Compiler output of SNMPv2-MIB specification file.

3. Ipstack MIB-II: The SNMP-Agent includes a generic implementation of the Ipstack MIB. You can use this as a basis for adding the support of your system Ipstack. In addition the Agent source code comes with Ipstack MIB-II implementation of many popular Ipstacks.

4. Adding Support for your MIB's: Additional MIB's can easily be added to the Agent as explained in the section "Adding Additional MIB's". The process involves compiling the ASN.1 files defining the MIB's, inserting user code in the appropriate sections within the produced 'C' files, and providing the API services as needed. Because DMH SNMP Agent supports dynamic registration of MIB objects, no change is required in the DMH SNMP kernel in order to support additional MIB's. Each MIB implementation files includes the code for dynamic registration of the objects implemented in that file.

5. Transport Services: The SNMP Agent requires transport services (typically datagram) to receive and send SNMP messages. Typically the transport is UDP/IP (the User Datagram Protocol, see [13]), using UDP port 161. Other transport protocols maybe used, although these are less common. DMH SNMP Agent is designed to be integrated and used with any transport services. This is explained in the section "Integrating with the Underlying Datagram Services".

6. MIB API (Application Interface): This is system specific code needed by method functions to get or set actual MIB objects. For example, in order to retrieve the object *sysUpTime,* DMH SNMP looks for an API function called *GetTimeTicks()* in the MIB API library provided by the hosting-system. In the section on "Implementing MIB-II objects" we detail how to implement the API module to fulfill the requirements of some MIB-II objects. API modules for additional MIB's are specific to the MIB application; however, the section on "Adding Additional MIB's" provides examples of such API modules.

7. Basic Hosting-System Services: These are services needed by the DMH SNMP kernel and the MIB modules regarding memory allocation, string formatting, memory copying, and console I/O for error reporting. Most systems already include services such as *malloc()* and *sprintf().*

Model of Operation

Following is a procedural description of the handling of an SNMP request from the moment it is received by the Agent, to the production and sending of a response to the requester.

1. **A Manager sends SNMP request**. An SNMP request PDU is made by a manager and sent to the IP address of the Agent network entity. By default, the destination port of the UDP datagram encapsulating the SNMP PDU is *SNMP_AGENT_PORT* (161).

2. **Agent receives SNMP Request**. The Agent receives the request (assume UDP datagram) and checks the UDP destination port. If the destination port is *SNMP_AGENT_PORT* (161) the entry point function that processes the inbound SNMP request, "*SnRcvDatagram* ()" is called. This function will process the SNMP request, build an SNMP reply and send it back to the requester. (Other UDP ports represent other applications such as TFTP, BOOTP, etc.).

3. **ASN.1/BER Parsing**. The ASN.1/BER parser parsers the encoded request and creates an SNMP PDU with a Variables link list in actual representation.

4. **Search the MIB tree.** DMH SNMP maintains a MIB data structure with information on every SNMP object maintained by the system (e.g. registered to DMH SNMP). Such information includes the object name, type, associated access function, optional instance function, and the like. The DMH SNMP kernel now searches the MIB tree to find an entry for the requested object (if this is a get-next request, the 'next' object is retrieved, or, for tables, a similar object with a 'next' instance value is retrieved).

5. **Evaluate the Instance.** This step is performed for tabular objects only: tabular objects are conceptual collections of rows in a table, which are addressed by a part of the SNMP variable name called instance. For each such object, DMH SNMP calls an "instance evaluator" routine associated with that object, which searches application tables for a row matching a given input instance (or the next one in case of Get-Next commands). The information is then maintained in the process scope to later be used by the object access routines (see below). Note that if several objects are requested within the same row of a given table, the instance function for that row is called only once, saving the system unnecessary work.

6. **Object Access**. Depending on the command a read or write operation is performed:

   - For Get and Get-Next commands, the access routine returns the data representing the object value at this time, possibly using instance information evaluated by the instance function. The retrieved data is encoded into a response PDU.

   - For Set commands, the access routine is called twice, first to test if a given input value is acceptable, and then to commit (e.g., to actually set an internal object to a given input value). In both cases instance information evaluated by the instance function might be used.

7. **ASN.1/BER Build**. The actual representation of the SNMP PDU consisting of the header and the VarBind List is now encoded back to ASN.1/BER machine independent representation.

   - In case of a Get PDU, the response contains the retrieved data with the original object names.

   - In case of a Get-Next PDU, the response contains the retrieved data with the updated object names reflecting the 'next' object relative to the original input name.

   - In case of a Set PDU, the response is contains the original names and data.

   - In case of a processing error (if a response is at all generated), the response contains the original names and data, with an error code indicating what the error was (e.g., no such object...).

8. **Send the reply**. Finally the encoded ASN.1/BER data is encapsulated in a transport (e.g. UDP) datagram. The destination port is set to the UDP source port of the request. The UDP datagram is sent to the IP address of the requester.

Integrating the DMH SNMP Agent

In order to utilize the DMH SNMP portable agent, it is required to make the agent aware of the actual system in which it is working. For example, different systems have different ways of calculating the current time or sending a datagram to the network. While some environments have standardized on an operating system such as UNIX, many developers are confronted with the need to support SNMP on less standardized embedded systems.

The integration process is based on a set of system specific services to be provided by the hosting system. These services are often available already, perhaps under a different name or syntax. Examples of such services are provided below. To facilitate the writing of the integration services, the DMH SNMP agent tool-kit includes 'stub' functions for every required system-specific service, with detailed comments explaining the expected integration code.

The process of integrating the DMH SNMP agent to most hosting systems is expected to last 1-2 weeks. In particular, if the optional DMH Software UDP/IP module is licensed along with DMH SNMP, you can expect to complete the whole integration in under a week.

At the end of the integration process, <u>**your system will have full SNMP capability and MIB-II support**</u>, meaning that users can use an SNMP manager to monitor basic parameters of your devices.

System Services

The following is a partial list of the main system services required by the DMH SNMP Agent. A full specification of each system service can be found in the DMH SNMP programmer's Manual.

<u>Datagram Services</u>

- ***SnGetXmtBuffer*** - Allocates a buffer suitable to data transmission.
- ***SnFreeBuffer*** - Frees a transmission buffer.
- ***SnXmtDatagram*** - Transmits a datagram via UDP/IP.
- ***SnRcvDatagram*** - An DMH SNMP function to be called when a datagram addressed to the SNMP agent is received

<u>MIB-II related services</u>

- ***SnGetSysOid*** - Returns the object ID of the current system.
- ***SnGetSysDescr***- Returns a string describing the current system.
- ***SnGetSysContact, SnSetSysContact*** - Reads/Sets a string representing a user defined system contact.
- ***SnGetSysName, SnSetSysName*** - Reads/Sets a string representing a user defined system name.
- ***SnGetSysLocation, SnSetSysLocation*** - Reads/Sets a string representing a user defined system location.
- ***SnGetSysServices*** - Returns a value indicating the type of services this system implements, e.g., repeater, router, etc.
- ***SnGetIfNum*** - Returns the number of communication interfaces in the system.
- ***SnGetIfStats*** - Retrieves a structure consisting of current interface statistics for a given interface, such as number of packets received, octets transmitted, errors, and the like.
- ***SnSetIfAdminStatus*** - Forces an interface to go down or up.

<u>Miscellaneous Services</u>

- ***GetTimeTicks*** - Returns the time, in 1/100 second, since the system started.
- ***consoleMsg*** - Reports a message to a local console, for error reporting, tracing, and so on.

MIB Development.

As implied in the architecture description, the MIB's are not strictly part of the DMH SNMP kernel. Rather, code implementing diverse MIB's can be added at any time.

Perhaps the main metric in the evaluation of an SNMP tool kit ought to be the ease of MIB addition - the reason is simple: while porting the basic kernel is done just once in the life of a project, adding MIB's is a constantly evolving process, as systems evolve over time with added functionality, which invariantly implies added MIB extensions.

The process of MIB addition is described below.

1. Identify a feature or related features of your system that should be managed by SNMP. Examples: Bridge, UPS, and Printer. If such feature(s) are not implemented, implement them or specify their implementation.

2. Define and implement a set of Application Interface (API) that the MIB module will use to access the snmp managed of feature(s).

3. If possible use an already predefined industry standard mib. Examples: BRIDGE-MIB, UPS-MIB, and Printer-MIB.

4. If required, write a Management Information Base (MIB) that specifies the objects, tables, indices, operations, etc., for the managed feature(s). Note that you can use part of the many standard MIB files already defined. (DMH MIB-Compiler comes with pre-defined industry MIBs).

5. Compile the MIB file using the DMH SNMP MIB compiler. The result is set 'C' and 'H' files as described below.

6. The next step involves adding application specific code to the compiler produced 'C' files. To facilitate this, the compiler inserts comments as shown where user code is required:

   ```
   /* start user code */
   /* end user code */
   ```

   In addition, the MIB-Compiler will generate notes for the implementer. Example:

   ```
   /* User-note: assign actual values returned by sysget_<function> call: */
   ```

DMH SMIv2 MIB-Compiler

What is the MIB-Compiler?

DMH SMIv2 MIB-Compiler is a tool to verify the correctness of a MIB file specified in the standard ASN.1 format and generate output files of various formats. The output for the DMH Advanced SNMP Agent is a set of 'C' and 'H' of the base-code and definitions for method functions and system functions required by the DMH Snmp-Agent engine.

The MIB-Compiler supports both SMIv2 and SMIv1. It can compile standard published IETF and IANA ASN.1 mib files. We have tested most the IETF and IANA MIBs. There is no need for a special directive as required by some other mib-compilers.

The front-end compiler is based on libsmi, an advanced research mib-compiler and mib processing related tool-set. The compiler includes several additional built-in back-ends for various formats including SMIng, SMIv1, SMIv2, import, type-query and more.

The MIB-Compiler Advantages:

- ✓ **Standard MIB files:** Can process standard ASN.1 MIB file format. No special directives are needed.
- ✓ **SMI Formats:** Both SMIv1 and SMIv2 ASN.1 formats are supported.
- ✓ **Command-line** tool, can be included in build tools (makefile etc.)
- ✓ **Most of the Code:** Generates most of the base code required by the DMH SNMP-Agent to add the support for given MIB.
- ✓ **Inline Comments for User:** Provides hints and comments in the generated output files to help adding system specific code.
- ✓ **Well Maintained and updated** to support the latest features and improvements in the DMH SNMP-Agent.
- ✓ **ASN.1 Linter:** Lint Includes ***smilint*** tool to verify MIB SMI syntax and rules.
- ✓ **Output Formats:** Generates other output formats in addition to DMH SNMP-Agent files.
- ✓ **Command line options:** Many command-line options to control the output 'C' files based on the style of the actual MIB implementation.
- ✓ **Services to customers:** generate custom formats, changes to meet special requirements.
- ✓ **New feature** Java MIB output for DMH Java SNMP Agent.

Output Files for the DMH Advanced SNMP Agent:

To generate code for DMH snmp-agent use command-line option ***-f dmhsnmp.***
The complier generates a set of source files for the compiled MIB file.

Example:

```
> smidump -c smi.cfg -f dmhsnmp ../mibs/ietf/IP-MIB

ip-mib.h created
ip-mib.c created
ip-mib-sys.c created

>
```

The files generated by the MIB-Compiler include MIB related definitions and structure in an 'h' file, snmp-agent specific 'c' file with method functions, and a system specific 'c' file where you add your code. The generated files are:

- `xxx-mib.c` - snmp specific code to access mib objects
- `xxx-mib-sys.c` - hosting system specific code
- `xxx-mib.h` - mib related structures and system function prototypes.

The MIB Structures and API function prototype file: `xxx-mib.h`

The mib header ('h') file is the API between the SNMP-Agent and your system specific functions. It includes definitions and structures for the compiled MIB, for both scalar and tabular mib objects. It also includes API function-prototypes for system specific get functions such as sysget_<function>, sysalloc_<ent>, sysget_xxx_scalars etc.

Both the main "c" file `xxx-mib.c` and the system file `xxx-mib-sys.c` include the header file `xxx-mib.h`.

The mib developer should review the generated code; add the correct dimension to arrays and the system specific fields if required.

The SNMP-Agent file: `xxx-mib.c`

This file is the main generated "c" mib file. The `xxx-mib.c` file contains snmp-agent specific code:

1) scalar objects method functions (f_<func>).
2) tabular object method functions and related structures. (i,b,i2e_<fnc>)
3) trap functions (trap_<function>) , and ;
4) mib database - a mib registration table, external initialization functions to register mibs.

The snmp-agent engine calls method functions in this file. The snmp method functions include calls to the system specific functions in `xxx-mib-sys.c`. The mib developer should review the generated code and add support code in table functions as well as object method functions.

The Hosting-System file: `xxx-mib-sys.c`

This file is the hosting-system (your system) "c" file generated by mib compiler. External functions in this file provide the access to scalar and tabular mib objects. For read-write/create tables there are functions to support entry manipulation (alloc/free functions).

Example for access functions:

- `sysget_xxx_scalars()` to get the actual scalar mib object values.
- `sysget_xxxEntry()` to get the actual tabular mib object values.
- `sysalloc_xxxEntry()` to allocate a new entry.
- `sysfree_xxxEntry()` to delete an existing entry.

This is a system specific file that "belongs" to your system. Functions in this file are called from the snmp file: `xxx-mib.c` to get or set actual objects.

The mib developer should review the generated stub-code and add code to provide access to actual tables and scalars. The actual mib table or scalar is implementation specific.

You may choose to use part of the code the mib-compiler generated. The mib-compiler has a command-line option to generate static data for tables. You can use those tables as a basis for your implementation or for development of a stand-alone agent for testing.

Example: The LED Project.

The following example should give you a feeling of the whole process.

Let's assume that our system includes a series of LED's, which we decided to monitor and control via SNMP. Assume that the LED's have three possible states: Off, Red, and Green. Assume the LED's and related hardware and software already exist in the system. Furthermore, assume the number of LED's, although currently limited to 4, might change in future versions of the product.

The MIB-Compiler generates the data structures for a Led entries, Scalar objects (one structure for each group) and "dummy" static tables and scalars structures. In this project we will use those tables as a basis for the implementation.

We will need to add code to the following generated function in `led-mib-sys.c` file:

- A routine to retrieve the LED scalars (actual number of LED's).
- A routine to retrieve an exact entry by its index (a static helper function).
- A routine to "walk" entries in the table or get an exact LED entry.

The functions are:

```
/**
 * sysget_led_scalars()
 *
 * system function for led scalar objects of samples
 **/
led_scalars_t *sysget_led_scalars(void);

/**
 * getexact_ledEntry()
 *
 * internal function to get exact ledEntry, a row of ledTable
 *
 **/
static
ledEntry_t *                          /* found entry, NULL if not found */
getexact_ledEntry(ledEntry_t *entry);

/**
 * sysget_ledEntry()
 *
 * system function for ledEntry, a row of ledTable
 *
 * The led table implementation is that the table is always lexi-sorted by
 * ledIndex. The get-next returns the first entry of index greater than the
 * input index.
 *
 **/
ledEntry_t *sysget_ledEntry(tableGetMode_t mode, ledEntry_t *entry);
```

The LED MIB is described below. We define a single scalar object for retrieval of "*NLeds*", and a table defining the state of each LED. The table will be indexed by *ledIndex*. Note that this MIB includes elements of varied complexity: a scalar read-only variable, which is the easiest to implement; a table of related elements, which requires the complexity of table -and instance- manipulation, and read-write objects, which require code for parameter validation and actual data writing.

```
-- the LED sample MIB

   led  OBJECT IDENTIFIER ::= { samples 1 }

nLeds OBJECT-TYPE
    SYNTAX      Integer32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION "The number of LED's in the system."
    ::= { led 1 }
```

```
-- the LED's table

-- The LED's table contains information on the entity's
-- LED's.

ledTable        OBJECT-TYPE
    SYNTAX      SEQUENCE OF LedEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION "A list of LED entries.  The number of
                 entries is given by the value of nLeds.
                "
    ::= { led 2 }

ledEntry        OBJECT-TYPE
    SYNTAX      LedEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION "An LED entry containing LED objects for a
                 particular LED.
                "
    INDEX  { ledIndex }
    ::= { ledTable 1 }

LedEntry ::=
    SEQUENCE {
      ledIndex       INTEGER,
      ledState       INTEGER,
      ledStorageType StorageType,
      ledStatus      RowStatus
    }

ledIndex        OBJECT-TYPE
    SYNTAX      Integer32 (1..8)
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION "A unique value for each LED. Its value
                 ranges between 1 and the value of nLeds.
                "
    ::= { ledEntry 1 }

ledState        OBJECT-TYPE
    SYNTAX      INTEGER {
                  off(1),   -- LED is off
                  red(2),   -- LED is on and red
                  green(3)  -- LED is on and green
    }
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION "The state of a LED. Setting this object to a
                 new value has the effect of turning an LED off,
                 on (green) or on (red)
                "
    ::= { ledEntry 2 }

ledStorageType   OBJECT-TYPE
    SYNTAX       StorageType
    MAX-ACCESS   read-create
    STATUS       current
    DESCRIPTION "The storage type for this conceptual row.
                 Conceptual rows having the value 'permanent' must
                 allow write-access at a minimum to ledState
                "
    DEFVAL      { nonVolatile }
    ::= { ledEntry 3 }

ledStatus        OBJECT-TYPE
```

```
    SYNTAX       RowStatus
    MAX-ACCESS   read-create
    STATUS       current
    DESCRIPTION "The status of this conceptual row.

                 Until instances of all corresponding columns are
                 appropriately configured, the value of the
                 corresponding instance of the ledStatus column
                 is 'notReady'.

                 In particular, a newly created row cannot be made active until
                 the corresponding ledState has been set.

                 The RowStatus TC [RFC2579] requires that this
                 DESCRIPTION clause states under which circumstances
                 other objects in this row can be modified:

                 The value of this object has no effect on whether
                 other objects in this conceptual row can be modified,
                 except for ledState. For this object, the
                 value of ledStatus must not be active.
                "
    ::= { ledEntry 4 }
```

To compile the MIB file, type:

```
> smidump –c smi.cfg -f dmhsnmp mbled.mib
```

If no errors are reported, the compiler generates the following files:

- `led-mib-sys.c` – system specific functions
- `led-mib.c` – snmp agent specific file
- `led-mib.h` – definition of scalars and table structures

The `led-mib-sys.c` includes the system specific functions. Let's look at the added code to the generated functions described above. A solid line on the left marks the added code.

The scalar structure and the get functions are:

```
/*
 * Static structure to hold led scalar mibs used by  sysget_led_scalars() below
 */
static led_scalars_t led_scalars = {

     0,                          /* nLeds int */
};

/**
 * sysget_led_scalars()
 *
 * system function for led scalar objects of samples
 **/
led_scalars_t *
sysget_led_scalars(void)
{
     /* static struct above is returned
      */
     return &led_scalars;
} /* end sysget_led_scalars() */
```

The LED table itself is a static array of LED entries. The initial table has one entry. Entries can be added or deleted by the RowStatus manipulation.

```
/*
 * ledTable
 *
 * Static array (table) of entries to keep actual data used by
 * sysget_ledEntry() below
 */
static ledEntry_t ledTable[MAX_LED_ENTRIES] = {

     {
          /* index components */
          1,                            /* ledIndex int */
          /* other components */
          1,                            /* ledState int */
          2,                            /* ledStorageType StorageType_e */
          0,                            /* ledStatus RowStatus_e */
     },
};

#define ledTable_size sizeof(ledTable)/sizeof(ledEntry_t)
```

The helper function to get an exact entry is used by the API function below:

```
/**
 * getexact_ledEntry()
 *
 * internal function to get exact ledEntry, a row of ledTable
 **/
static
ledEntry_t *                            /* found entry, NULL if not found */
getexact_ledEntry(ledEntry_t *entry)
{
    int i;
    int aindex = entry->ledIndex - 1; /* array index */

    if (aindex < 0 || aindex > ledTable_size-1) {
        return NULL;            /* array index is out of range */
    }

    for (i = 0 ; i < ledTable_size ; i++) {

        if (ledTable[i].ledStatus == RS_free)
            continue;           /* skip free entry */

        if (ledTable[i].ledIndex == entry->ledIndex)
            return &ledTable[i]; /* found exact entry */
    }

    return NULL;   /* no entry found */

} /* end getexact_ledEntry() */
```

The API function to walk the table or get an exact entry:

```
/**
 * sysget_ledEntry()
 *
 * system function for ledEntry, a row of ledTable
 *
 * The led table implementation is that the table is always lexi-sorted by
 * ledIndex. The get-next returns the first entry of index greater than the
 * input index.
 *
 **/
ledEntry_t *
sysget_ledEntry(tableGetMode_t mode, ledEntry_t *entry)
{
     int i;

     if (entry->ledIndex < 0 || entry->ledIndex >= ledTable_size) {
          warningMsg("sysget_ledEntry: ldIndex %d out of range", entry->ledIndex);
          return NULL;
     }

     if (mode == tableGetExact) {
          /* find and return exact entry */
          return getexact_ledEntry(entry);
     }
     else if (mode == tableGetFirst) {

          /* find and return first entry
           */
          for (i = 0 ; i < ledTable_size ; i++) {

               if (ledTable[i].ledStatus == RS_free)
                    continue;           /* skip free entry */

               return &ledTable[i]; /* first entry */
          }

          return NULL;   /* no entry found */
     }
     else if (mode == tableGetNext) {

          /* return next-lex entry with respect to input index (of snmp request)
           */
          int ledindex = entry->ledIndex - 1;

          for (i = ledindex ; i < ledTable_size ; i++) {

               if (ledTable[i].ledStatus == RS_free)
                    continue;           /* skip free entry */

               if (ledTable[i].ledIndex > entry->ledIndex)
                    return &ledTable[i]; /* found greater (nex-lex) entry */
          }

          return NULL;   /* no entry found */

     }
     else {
          return NULL;   /* bad table mode */
     }

     return NULL;

} /* end sysget_ledEntry() */
```