SNMP User Guide

Version 1.00

For use with Simple Network Management Protocol (SNMP) modules versions 3.04 and above

Exported on 02/05/2019

All rights reserved. This document and the associated software are the sole property of HCC Embedded. Reproduction or duplication by any means of any portion of this document without the prior written consent of HCC Embedded is expressly forbidden.

HCC Embedded reserves the right to make changes to this document and to the related software at any time and without notice. The information in this document has been carefully checked for its accuracy; however, HCC Embedded makes no warranty relating to the correctness of this document.

Table of Contents

1 System Overview 6
1.1 Introduction 7
1.2 Feature Check 8
1.3 Packages and Documents 9
Packages 9
Documents 10
1.4 Change History 11
2 Source File List 12
2.1 API Header Files 12
2.2 Configuration Files 12
2.3 Version Files 13
2.4 System Files 13
SNMP Base System Files 13
SNMP Manager System Files 13
SNMP Agent System Files 14
SNMPv3 System Files 14
2.5 MIB Compiler Files 15
3 Configuration Options 16
3.1 config_ip_app_snmp.h 16
Common Options 16
System Information MIB Options 17
Manager Options 18
SNMPv3 Options 19
3.2 config_ip_app_snmp.c 20
4 The MIB Compiler 22
4.1 Running the Compiler 23
4.2 MIB Compiler Input 23
4.3 Output Files 25
snmpvars.c 26
The .h file 27
The .c file 28
The .num file 29

Organizing the Output Files....29
Suggested Data Structures....30
C Routine Frames....33
4.4 Validation of Objects Defined in the MIB....35
5 Application Programming Interface....37
5.1 Module Management....37
snmp_base_init....38
snmp_base_register_encryption....39
snmp_base_register_hash....40
snmp_register_file_ifc....41
snmp_register_mib_ifc....42
snmp_base_start....43
snmp_base_stop....44
snmp_base_delete....45
5.2 SNMP Manager....46
Manager Functions....47
snmp_connect....48
snmp_disconnect....49
snmp_get_req....50
snmp_set_req....51
snmp_get_next_req....52
snmp_get_bulk_req....53
snmp_recv....54
snmp_release_buf....55
Utility Functions....56
snmp_get_varbind....57
snmp_obj_rd_int32....58
snmp_obj_rd_int64....59
snmp_obj_rd_uint32....60
snmp_obj_wr_int32....61
snmp_obj_wr_int64....62
snmp_obj_wr_uint32....63
snmp_write_oid....64
snmp_write_oid_obj....65
snmp_get_snmp_mib2_stats....66
Manager Callback Functions....67
t_snmp_conn_cb....68

t_snmp_ntf_cb 69
5.3 SNMP Agent 70
File Interface Functions 71
t_snmp_get 72
t_snmp_get_next 73
t_snmp_set 74
Agent Functions 75
snmp_send_ntf 76
snmp_reg_cb 77
Agent Callback Functions 78
t_snmp_open_cb 79
t_snmp_close_cb 80
t_snmp_read_cb 81
t_snmp_write_cb 82
t_snmp_seek_cb 83
5.4 Error Codes 84
5.5 Error Status Values 85
5.6 MIB Status Values 86
5.7 Types and Definitions 87
PDU Types 87
Connection Events 88
t_snmp_mib_obj 88
t_snmp_oid 88
t_snmp_encr_par 89
t_mib2_snmp 89
t_snmp_file_mode 89
t_snmp_file_ifc_dsc 90
t_snmp_mib_ifc_dsc 90
t_snmp_auth and t_snmp_encr 91
t_snmp_community 91
Access Rights 92
t_snmp_conn_conf 92
t_snmp_user 93
t_snmp_ntf 93
Notification Types 94
t_snmp_trap 94
SNMP Manager Connection States 95
SNMP Version Numbers 95

Object Types ... 96

6 SNMP Manager Code Example ... 97

7 Integration ... 101

7.1 OS Abstraction Layer ... 101

7.2 Utilities ... 101

7.3 PSP Porting ... 102

psp_get_tick_count ... 103

psp_getrand ... 104

1 System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

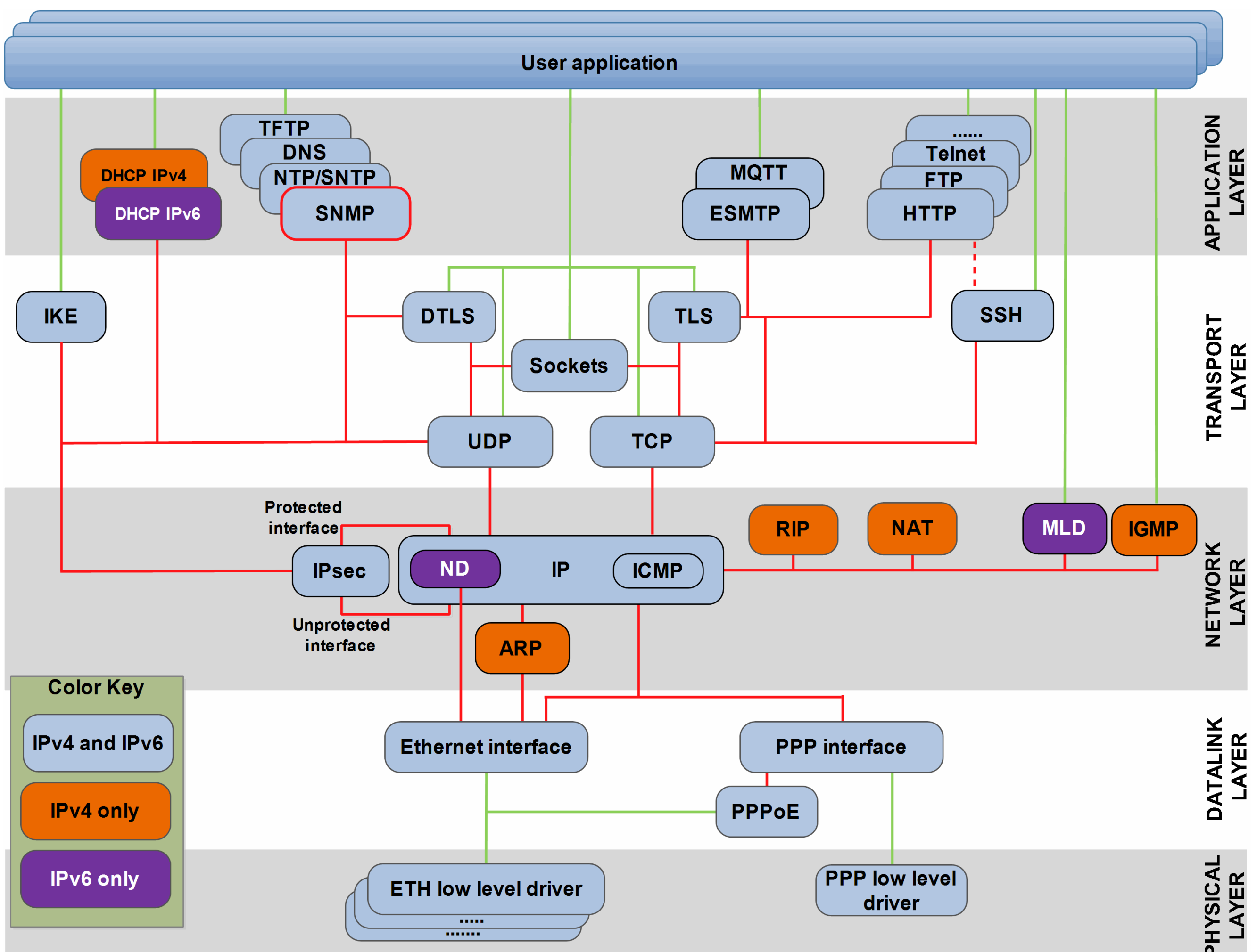
- Introduction – describes the main elements of the module. This section includes a diagram showing the position of this module within HCC's TCP/IP stack.
- Feature Check – summarizes the main features of the module as bullet points.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

1.1 Introduction

This guide is for those who want to implement one or more of HCC Embedded's SNMP modules: Base, Manager, Agent and/or SNMPv3. Users can use any of the modules on its own, or combine them on a network for a complete SNMP solution.

The Simple Network Management Protocol (SNMP) is used to manage devices on IP networks. Devices running an SNMP agent application report events and traps to a computer acting as an SNMP manager, running an SNMP manager application. One or more Management Information Bases (MIBs) are used to store management data and users can provide their own MIBs. The provided MIB compiler supports the MIB2 and SNMP MIBs and allows you to build MIBs.

The SNMP modules are part of the HCC MISRA-compliant TCP/IP stack, as shown below. They are designed specifically for use with this TCP/IP stack. (In this diagram green lines show interfaces available to users of the stack, red lines show interfaces internal to the TCP/IP system.)



The SNMP modules use a User-based Security Model (USM). This incorporates:

- Authentication using Message Digest 5 (MD5) and the Secure Hash Algorithm 1 (SHA-1).
- Data encryption using Triple Data Encryption Standard (3DES).

1.2 Feature Check

The main features of the system are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Complies with HCC's MISRA-compliant TCP/IP stack.
- Designed for integration with both RTOS and non-RTOS based systems.
- SNMP implementation (RFC 1157) supports SNMPv2c (RFC 1901) and SNMPv3 (RFC 3410).
- A MIB compiler is included. The MIB implementation is compliant with RFC 3413 and RFC 3418.
- View-based Access Control Model (VACM) and the VACM Management Information Base (MIB) are compliant with RFC 3415.
- User-based Security Model (USM) is compliant with RFC 3414.
- Authentication using Message Digest 5 (MD5) and the Secure Hash Algorithm 1 (SHA-1).
- Data encryption using the Triple Data Encryption Standard (3DES).

1.3 Packages and Documents

Packages

The following table lists the packages that you need in order to use this module:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
ip_app_snmp_base	The SNMP base package (documented in this guide). This is required by both SNMP Manager and SNMP Agent.
ip_app_snmp_core_v3	The SNMP version 3 package, required if SNMP V3 is used. (This is documented in this guide).
ip_app_snmp_manager	The SNMP Manager package, if required. (This is documented in this guide).
ip_app_snmp_agent	The SNMP Agent package, if required. (This is documented in this guide).
ip_app_snmp_agent_demo	The SNMP Agent demo package, if required.
ip_scad_base	The Socket Adaptor (SCAD) base package.
psp_template_base	The Platform Support Package (PSP).
oal_base	The OS Abstraction Layer (OAL) package.
mutil_timer	The MISRA-compliant timer utility.
util_lib_ber	The library that handles Basic Encoding Rules (BER) encoding.
enc_base	The Embedded Encryption Manager (EEM) package (only if secure or authorized connection is needed).
ip_app_snmp_comp	The MIB Interface Adapter for MIB Compiler package.
ip_app_snmp_comp_demo	The example MIB tree to use with the above adapter package.

Documents

For an overview of the HCC TCP/IP stack software, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC TCP/IP Dual Stack System User Guide

This is the core document that describes the complete TCP/IP stack. It covers both IPv4 and IPv6 systems.

HCC Embedded Encryption Manager User Guide

This manual documents the EEM, which provides the handles for the hash and encryption algorithms.

HCC SNMP User Guide

This manual documents the SNMP Base, Manager, and Agent modules.

1.4 Change History

This section describes past changes to this manual.

- To download manuals, see TCP/IP PDFs.
- For the history of changes made to the package code itself, see History: ip_app_snmp_base, History: ip_app_snmp_agent and History: ip_app_snmp_manager.

The current version of this manual is 1.00.

Manual version	Date	Software version	Reason for change
1.00	2019-02-05	3.04	First release.

2 Source File List

The following sections describe all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the HCC Source Tree Guide. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

Note: Do not modify any files except the configuration files.

2.1 API Header Files

The Application Programming Interface (API) header files are the only files that should be included by an application using this module.

File	Package	Description
src/api/api_ip_app_snmp.h	**ip_app_snmp_base**	Module Management functions.
src/api/api_ip_app_snmp_man.h	**ip_app_snmp_manager**	Manager functions.
src/api/api_ip_app_snmp_agent.h	**ip_app_snmp_agent**	Agent functions.

2.2 Configuration Files

Configure these files in the directory **src/config** as required. These are the only files in the module that you should modify.

File	Package	Description
config_ip_app_snmp.h	**ip_app_snmp_base**	Contains all the configurable SNMP parameters. For details, see Configuration Options.
config_ip_app_snmp.c	**ip_app_snmp_agent**	Contains arrays used for community-based authentication, SNMPv3 Agent Engine IDs, and the security model.

2.3 Version Files

These files in directory **src/version** of each package contain the version number of the module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

File	Package	Description
ver_ip_app_snmp.h	**ip_app_snmp_base**	Base version.
ver_ip_app_snmp_man.h	**ip_app_snmp_manager**	Manager version.
ver_ip_app_snmp_agent.h	**ip_app_snmp_agent**	Agent version.
ver_ip_app_snmp_v3.h	**ip_app_snmp_v3**	SNMPv3 package version.

2.4 System Files

SNMP Base System Files

These files are in the directory **src/ip/apps/snmp** in each package. **These files should only be modified by HCC**.

File	Description
snmp.h	Main SNMP header file.
snmp_base.c	Main SNMP source code.
snmp_pdu.c and **.h**	PDU source code and header file.
snmp_util.c	Utility functions source code.

SNMP Manager System Files

These files are in the directory **src/ip/apps/snmp** in the **ip_app_snmp_manager** package. **These files should only be modified by HCC**.

File	Description
snmp_man.c	Manager functions source code.
snmp_man.h	Manager functions header file.

Copyright HCC Embedded 2019

SNMP Agent System Files

These files are in the directory **src/ip/apps/snmp** in the **ip_app_snmp_agent** package. **These files should only be modified by HCC**.

File	Description
snmp_ag.c and **.h**	Agent functions source code and header file.
snmp_mib.c and **.h**	Internal MIB source code and header file.
snmp_ntf.c and **.h**	Notification mechanism source code and header file.
snmp_usm_ag.c and **.h**	Agent USM functions source code and header file.

SNMPv3 System Files

These files are in the directory **src/ip/apps/snmp** in the **ip_app_snmp_v3** package. **These files should only be modified by HCC**.

File	Description
snmp_usm.c and **.h**	User-based Security Model (USM) source code and header file.
snmp_v3.c and **.h**	SNMPv3 source code and header file.
snmp_v3_types.h	Header file with SNMPv3 and USM parameters.

Copyright HCC Embedded 2019

2.5 MIB Compiler Files

These files are in the directory **hcc/util/snmp**.

File	Description
mibcomp.exe	DOS executable.
rfc1213.mib	MIB II definition file.
rfc2011.mib	MIB definition file for IP and ICMP.
rfc2012.mib	MIB definition file for TCP.
rfc2013.mib	MIB definition file for UDP.
rfc2571.mib	MIB definition file for SNMP.
rfc2572.mib	MIB definition file for Message Processing and Dispatching.
rfc2573.mib	MIB definition file for SNMP.
rfc2574.mib	MIB definition file for SNMP User-based Security Model.
rfc2575.mib	MIB definition file for SNMP View-based Access Control Model.
rfc2578.mib	MIB definition file for SNMPv2-SMI.
rfc2579.mib	MIB definition file for SNMPv2-TC.
rfc3411.mib	MIB definition file for SNMP Management Architecture.
rfc3412.mib	MIB definition file for Message Processing and Dispatching.
rfc3413.mib	MIB definition file for SNMP.
rfc3413a.mib	MIB definition file for SNMP notifications.
rfc3414.mib	MIB definition file for User-based Security Model.
rfc3415.mib	MIB definition file for View-based Access Control Model.
rfc3418.mib	MIB definition file for SNMP entities.
rfc3584.mib	MIB definition file for coexistence between SNMPv1, v2c, and v3.

3 Configuration Options

Set the configuration options in these files:

- **src/config/config_ip_app_snmp.h** in the base package – contains the configuration options.
- **src/config/config_ip_app_snmp.c** in the agent package – contains arrays used for community-based authentication, SNMPv3 Agent Engine IDs, and the security model.

3.1 config_ip_app_snmp.h

Set the configuration options in the base package's **src/config/config_ip_app_snmp.h** file. This section lists the options and their default values.

Common Options

SNMP_CLIENT_TASK_STACK_SIZE

The size of the SNMP client task stack. The default value is 1024.

SNMP_MAX_CONN

The maximum number of SNMP connections that can be open simultaneously. The default value is 2.

SNMP_TIMER_PERIOD

The timer period used by the module in ms. The default value is 10.

SNMP_CONNECTION_TIMEOUT

The connection timeout in ms. The default value is 1000.

SNMP_NTF_TARGET_NR

The number of notify targets that can be registered in the SNMP module. The default value is 1.

SNMP_COMMUNITY_NR

The number of SNMP agent users (community profiles). The default value is 1.

SNMP_MIB2_SNMP_ENABLE

Set this to 1 to enable the SNMP statistics MIB. The default value is 0.

Copyright HCC Embedded 2019

SNMP_MAX_VARBINDINGS

The maximum number of variable bindings that can be handled by the SMTP module. The default is 10.

This is the size of the internal table that is used for handling variable binding requests. The maximum (theoretical) value for a packet is:

(MTU size - (SNMP header (about 23) + IP hdr + UDP hdr + ETH hdr)) / 7.

SNMP_MAX_OID_LENGTH

The maximum length of the buffer holding an Object ID (OID). The default value is 20.

SNMP_MAX_DATA_LENGTH

The maximum length of the buffer used for obtaining object data. The default value is 40.

SNMP_AGENT_ENABLE

Set this to 1 to enable operation of the SNMP agent. The default value is 0.

SNMP_LISTEN_PORT

The port that the agent listens on . The default value is 161.

SNMP_TRAPLISTN_ENABLE

Set this to 1 to enable the SNMP Trap Listener. The default value is 0.

SNMP_NTF_LISTEN_PORT

The notification/trap listen port. The default value is 162.

SNMP_INT_MIB_EN

Set this to 1 to enable an internal MIB. The default value is 0.

System Information MIB Options

SNMP_MIB_EXT_CNT

The maximum number of MIB interfaces that can be registered. The default value is 1.

SNMP_MIB_SYS_NOID_LEN

The system MIB OID descriptor length. The default value is 8.

SNMP_MIB_SYS_DESCRIPTOR

The system MIB descriptor string. The default value is "www.hcc-embedded.com".

SNMP_MIB_SYS_DESCRIPTOR_LEN

The system MIB descriptor length. The default value is (sizeof (SNMP_MIB_SYS_DESCRIPTOR) - 1).

SNMP_MIB_SYS_CONTACT

The system MIB contact string. The default value is "Hcc coders".

SNMP_MIB_SYS_CONTACT_LEN

The length of the system MIB contact string. The default is (sizeof (SNMP_MIB_SYS_CONTACT) - 1).

SNMP_MIB_SYS_NAME

The system MIB name string. The default value is "SNMP base".

SNMP_MIB_SYS_NAME_LEN

The system MIB name length. The default value is (sizeof (SNMP_MIB_SYS_NAME) - 1).

SNMP_MIB_SYS_LOCATION

The system MIB location string. The default value is "Hungary".

SNMP_MIB_SYS_LOCATION_LEN

The system location length. The default value is (sizeof (SNMP_MIB_SYS_LOCATION) - 1).

Manager Options

SNMP_MAN_ENABLE

Set this to 1 to enable the manager functions. The default value is 0.

SNMP_CONTEXT_MAX_LEN

The maximum length of a context/community name passed to the manager function. The default value is 10.

SNMP_UNAME_MAX_LEN

The maximum length of a user name. The default value is 10.

SNMP_MAN_PORT

The manager port number. The default value is 60000.

SNMP_MAX_NTF_LISTN

The maximum number of notify listeners. The default value is 2.

SNMPv3 Options

SNMP_V3_ENABLE

Keep the default of 1 to enable SNMPv3.

> **Note: SNMP_V3_ENABLE** must be set for the following options to take effect.

SNMP_ENGINE_ID_MAX_LENGTH

The maximum engine ID length. The default value is 32.

SNMP_ENGINE_ID_LENGTH

The system engine ID length. The default value is 13.

SNMP_USER_NR

The maximum number of agent users. The default value is 1.

SNMP_TIME_WINDOW

The time window of a connection in seconds. The default value is 150.

SNMP_USM_AES_ENABLE

Set this to 1 to enable the AES extension. The default value is 0.

SNMP_HASH_ALG_NR

The number of hash algorithms. The default value is 1.

SNMP_ENCR_ALG_NR

The number of encryption algorithms. The default value is 1.

SNMP_USM_KEY_LENGTH

The maximum length of a USM user password/key. The default value is 16.

This is dependent on the hash mechanism used. Set this to 16 for MD5, 20 for SHA-1, or 32 for AES256.

SNMP_USM_IV_LENGTH

The maximum size of the USM initialization vector. Set this to 8 for DES or 16 for AES. The default value is 8.

Copyright HCC Embedded 2019

3.2 config_ip_app_snmp.c

Configure the arrays described below in the file **src/config/config_ip_app_snmp.c** in the agent package.

System OID

To translate the community-based authentication scheme to SNMPv3, the array *g_mib_sys_noid[]* is used:

```
/* System OID */
/* Example of CISCO3000 device UID */
uint8_t g_mib_sys_noid[SNMP_MIB_SYS_NOID_LEN] = {0x2B,0x06,0x01,0x04,0x01,0x09,0x1,
0x06};
```

Community

To translate the community-based authentication scheme to SNMPv3, the array *g_snmp_community[]* is used:

```
const t_snmp_community g_snmp_community[SNMP_COMMUNITY_NR] =
{
 { (uint8_t*)"public", 6U, 0x0A000000U, 0xFF000000U, SNMP_TFLAGS_ACCESS_RW }
};
```

SNMP Targets

Each SNMP Agent has targets. These are defined by the array *g_snmp_targets[]*:

```
const t_snmp_trap g_snmp_targets[SNMP_NTF_TARGET_NR] =
{
/* { SNMP_VER_SNMP2VC,"public",6U,0U,SNMP_NTF_TYPE_TRAP,0,0,{ 0xC0A80018, 1602} } */
  { SNMP_VER_SNMP2VC, (uint8_t*)"public", 6U, 0U, SNMP_NTF_TYPE_NOTIFICATION, 2, 100,
{ { {10u, 1u, 3u, 240u }, IPV_IP_V4 }, 1602} }
/* { SNMP_VER_SNMP3,NULL,0U,2U,SNMP_NTF_TYPE_NOTIFICATION,5,100, {0xC0A80018,1602} } */
};
```

Engine ID (SNMP v3 only)

Each SNMP Agent has a unique engine identifier. This is defined by the array *g_snmp_engine_id[]*:

```
/* Engine ID
 * Format of this parameter is described by RFC3411: snmpEngineID */
uint8_t g_snmp_engine_id[SNMP_ENGINE_ID_LENGTH] =
{
  0x80U, 0x00U, 0x00U, 0x01U, 0x80U, 0x01U, 0x02U, 0x03U,
  0x04U, 0x05U, 0x06U, 0x07U, 0x08U
};
```

Users (SNMP v3 only)

The SNMP agent uses a User-based Security Model (USM). Users are defined by name, password (at least 8 characters), authentication protocol, and encryption method. The authorization and encryption key is derived from the password by using a password-to-key algorithm.

This is defined by the array *g_snmp_users[]*:

```
const t_snmp_user g_snmp_users[SNMP_USER_NR] =
{
  { (uint8_t*)"user0",5U,NULL,0U,NULL,
0U,SNMP_AUTH_NONE,SNMP_ENCR_NONE,SNMP_COMM_ACCESS_RW }
};
```

In this example user0 uses no authentication or encryption.

4 The MIB Compiler

The MIB compiler, a DOS executable named **mibcomp.exe**, supports the MIB2 and SNMP MIBs.

It takes as input the SNMP MIB specification written in ASN.1 compliant notation and produces one or more of the following output files:

- C language source code files that are useful as a starting point for implementing SNMP MIBs in SNMP agents. These files contain skeleton routines that must be filled in by the porting developer.
- A variables file that contains a table linking MIB variables to the skeleton routines.
- **.h** files that contain the function prototypes and definitions for the above C files.
- A "numbers" file, **.num**, suitable for describing the MIB variables to an SNMP manager application.

For more details of these files, see Output Files.

Filling in the skeleton routines (also called stub routines) mentioned above is a major portion of the work involved in porting and maintaining your SNMP agent. The routines can be quite complex, so a major portion of this section is devoted to them. Generally the hardest part of an implementation is understanding what these routines do: how they index Object Ids and access variables in tables.

Note: You will probably use the MIB compiler regularly during the development of the SNMP agent as MIB variables are changed. Store the **mibcomp** executable in a directory where it can be invoked by the makefiles that build your embedded system.

4.1 Running the Compiler

This section shows how to run the MIB compiler.

- To display the full list of options shown below, enter "`mibcomp`". (Several of the options generate additional information; see the file **parse.h** for more information.)

```
usage: mibcomp -i [infile ...] -a [MIBdefinitionFile ...] [-cefhnprsv]
        -c will output variables as a C file structure
        -e will produce enumerated values from MIB (of type struct enumList)
        -f will make pointers 'far' for intel x86
        -h will produce ".h" file for -c option functions
        -n will produce a numbers file (default)
        -p will make sure the C code is "pre-ANSI" (simple prototypes, etc)
        -r will produce validation info (ranges) in SNMP variables table
        -v will produce SNMP variables table
```

- To compile a single MIB, enter the following command:

```
mibcomp -i rfc1213.mib -chvn
```

- To compile SMIv2 MIBs, compile RFC2578.MIB and RFC2579.MIB as well, like this:

```
mibcomp -i rfc1213.mib rfc2578.mib rfc2579.mib -chvn
```

- To compile multiple MIBs on a single line, enter as many MIBs as the DOS command line allows. For example:

```
mibcomp -i rfc1213.mib rfc2578.mib rfc2579.mib rfc3411.mib rfc3412.mib rfc2571.mib
rfc2572.mib rfc3413a.mib -chvn
```

- Another way to compile multiple MIB files in one command is to create a file containing a list of the MIB files, one file per line, and use the "-a" option. In the following example the file is named **miblist**. (The file **miblist** is available in the **mibcomp** directory.)

```
mibcomp -a miblst -chvn
```

4.2 MIB Compiler Input

The MIB Compiler takes as input one or more MIB definition files as described by RFC 1155. Samples of such files are shipped with the HCC SNMP package, as listed in the Source File List. These sample files were created using a simple text editor to edit out the non-ASN.1 portions for the RFCs each MIB came from. By deleting all the text preceding the `DEFINITIONS ::= BEGIN` line, all the text following the `END` line, and all the page break text (footer, page break, header), one is left with the ASN.1 which is suitable for input to the HCC compiler.

You can, of course, define your own MIBs. Deciding which variables you want to manage and how to organize them is the hardest task. Writing the ASN.1 text to describe them is relatively easy. Once the MIB has been written in RFC 1155-compliant ASN.1, it can be compiled and implemented like any standard MIB.

Note the following:

- There is no restriction on the order in which MIB objects are defined.
- MIBs must be compiled in the correct order as they are incrementally parsed. So, if one MIB refers to objects in another MIB, the second MIB must be compiled first. For example, RFC 2571 refers to object *snmpModules* from RFC 2578, so RFC 2578 must be compiled before RFC 2571.
- Different MIBs can have the same object names. Different MIBs can have different object names. And they can have different OIDs too. The MIB compiler handles this situation by incrementally parsing MIBs and creating the tree. If two objects (in different MIBs) have the same names and the same OID, the later definition is used.
- The MIB compiler has the macros found in RFC 2578 and RFC 2579 (shown below) built-in. The provided files **rfc2578.mib** and **rfc2579.mib** eliminate these macros and other pre-defined base and built-in ASN.1 types. The compiler does not support redefinition of these macros and built-in types so they must not appear in any **.mib** files.

```
SNMPv2-SMI DEFINITIONS ::= BEGIN
MODULE-IDENTITY MACRO
OBJECT-IDENTITY MACRO
NOTIFICATION-TYPE MACRO

SNMPv2-TC DEFINITIONS ::= BEGIN
TEXTUAL-CONVENTION MACRO
```

- The OBJECT-IDENTITY *zeroDotZero* is not currently handled properly and is commented out in the file **rfc2578.mib**. Do not use "zeroDotZero" in other mib files; replace it with { 0 0 } instead.
- The MIB compiler currently does not properly handle SPACE characters within a range statement. For example, "`1 .. 5`" will return an error; instead use "`1..5`".
- Some MIB files contain a list of values or ranges. If a list contains more than 10 values (or five ranges), increase the value of `#define MCT_MAX_VALUES` from 10 to the actual number needed (or more), otherwise an error will be returned.

Copyright HCC Embedded 2019

4.3 Output Files

The compiler produces the following files when the options `-chvn` are used together.

Note: Since these files are re-generated whenever the MIB compiler is run (whenever the MIB is changed), do not edit them.

File	Description
snmpvars.c	Contains a list of all the OIDs in the MIBs. Contains the table mapping the MIB variables and containing stubs for groups in all the parsed MIBs. This contains the *variables[]* structure that the SNMP agent code uses to associate a C routine with individual variables. More on the variables structure in the next section.
.h file	Include file with prototypes for the C stub routines, token definitions for the MIB variables, and suggested data structures to contain the variables for each group or sequence in the MIB.
.c file	Contains stubs for the C routines prototyped in the **.h** file (stub routines for accessing MIB variables). This contains stubs for groups in all the parsed MIBs. Copy these to your own source files and complete them there. These routines are framed and commented, but they are only empty frames with no internal code. The areas where code needs to be added to actually implement the variables are flagged with the text TODO (all in CAPS, as shown). Part of the work of implementing a new MIB is to replace the TODO lines with C code that performs the intended `SET` or `GET` operation and returns the correct values. More details on this are given below in the section on C Routine Frames.
.num file	Numbers file - contains all the MIB objects with their Object Ids (OIDs) in dot notation with the corresponding symbols and their data types.

The names of the **.c** and **.h** files are set by the first eight characters of the name given on the DEFINITIONS line in the last MIB file parsed. For example:

- RFC 1213's `DEFINITIONS` line is as follows. This results in a file name of **`rfc1213_.c`**. The names of the other output files are determined in the same way; in this case: **rfc1213_.h** and **rfc1213_.num**.

```
RFC1213-MIB DEFINITIONS ::= BEGIN
```

- If RFC2575.MIB is the last MIB and its first line is "`SNMP-VIEW-BASED-ACM-MIB DEFINITIONS ::= BEGIN`", then file `snmp-vie.c` will be generated.

snmpvars.c

This example shows the first and final lines of a *variables[]* table:

```
struct variable variables[] = {
  {{1,3,6,1,2,1,1,1,0},   9, STRING, SYSDESCR, RONLY, var_system },
  {{1,3,6,1,2,1,1,2,0},   9, OBJID, SYSOBJECTID, RONLY, var_system },
  {{1,3,6,1,2,1,1,3,0},   9, TIMETICKS, SYSUPTIME, RONLY, var_system },
....
  {{1,3,6,1,2,1,11,28,0}, 9, COUNTER, SNMPOUTGETRESPONSES, RONLY, var_snmp },
  {{1,3,6,1,2,1,11,29,0}, 9, COUNTER, SNMPOUTTRAPS, RONLY, var_snmp },
  {{1,3,6,1,2,1,11,30,0}, 9, INTEGER, SNMPENABLEAUTHENTRAPS, RWRITE, var_snmp },
};
```

For each variable this shows:

- OID
- Type - string, counter, integer, and so on.
- Name - SYSDESCR, SNMPOUTGETRESPONSES and SNMPENABLEAUTHENTRAPS are examples.
- The function - this is **var_system()** or **var_snmp()** in this example.

For more details, see The Variables Structure.

The .h file

This example shows one set of tokens from a **.h** file:

```
/* Tokens for passing to "var_" routines. These can
serve as offsets into the generated MIB group tables.
*/

/* tokens for 'system' group */
#define SYSDESCR     0
#define SYSOBJECTID  SYSDESCR+4
#define SYSUPTIME    SYSOBJECTID+4
#define SYSCONTACT   SYSUPTIME+4
#define SYSNAME      SYSCONTACT+4
#define SYSLOCATION  SYSNAME+4
#define SYSSERVICES  SYSLOCATION+4
```

This example shows two prototypes for "var_" routines from a **.h** file:

```
/* prototypes for "var_" routines in variables table.
Simply delete any that aren't used */

u_char *
var_system(struct variable * vp,
        oid* name, int * len, int exact,
        int * var_len);

u_char *
var_interfaces(struct variable * vp,
        oid* name, int * len, int exact,
        int * var_len);
```

This example shows a recommended MIB table from a **.h** file:

```
/* MIB table for 'system' group */

struct system_mib {
        void *   sysDescr;       /* 32 bit ptr */
        oid  *   sysObjectID;    /* 32 bit ptr */
        u_long   sysUpTime;
        void   * sysContact;     /* 32 bit ptr */
        void   * sysName;        /* 32 bit ptr */
        void   * sysLocation;    /* 32 bit ptr */
        long     sysServices;
};
```

Copyright HCC Embedded 2019

The .c file

This example shows one function from a **.c** file, showing a "TODO" area that requires hand editing:

```
u_char *
var_interfaces(
  struct variable * vp, /* IN - pointer to variables[] entry */
  oid * name,           /* IN/OUT - input name requested; output name found */
  int * length,         /* IN/OUT - length of input & output oids */
  int   oper,           /* IN - NEXT_OP (=0), GET_OP (=1), or SET_OP (=-1) */
  int * var_len)        /* OUT - length of variable, or 0 if function */
{

  /* TODO: Add code here*/

  USE_ARG(vp);
  USE_ARG(name);
  USE_ARG(length);
  USE_ARG(oper);
  USE_ARG(var_len);
  return NULL;         /* default FAIL return.*/
}
```

Copyright HCC Embedded 2019

The .num file

This example shows the first and final lines of a .num file showing its OID numbers:

```
Object Id numbers for MIB rfc1213_:

1                                iso
1.3                              org
1.3.6                            dod

...

1.3.6.1.2.1.11.28                snmpOutGetResponses       Counter
1.3.6.1.2.1.11.29                snmpOutTraps              Counter
1.3.6.1.2.1.11.30                snmpEnableAuthenTraps     INTEGER
1.3.6.1.3                        experimental
1.3.6.1.4                        private
1.3.6.1.4.1                      enterprises
```

Organizing the Output Files

We recommend placing the files **snmpvars.c** and **snmp-vie.h** in the SNMP directory and placing the implementations of functions (prototypes in **snmp-vie.c**) in implementation directories.

Example

Assume that RFC2578, RFC2579, RFC2571, RFC2572, RFC2573, RFC2574, and RFC2575 are to be added to the default **rfc1213.mib** and these will be in the **/snmpv3** directory. The following applies:

1. The compiled RFCs are **rfc1213.mib** (MIB-2), **rfc2578.mib** (SNMPv3 basic sub-trees), **rfc2579.mib** (textual-conventions for SMIv2), **rfc2571.mib**, **rfc2572.mib**, **rfc2573.mib**, **rfc2574.mib**, and **rfc2575.mib**.
2. The MIB compiler generates the files **smnp-vie.c**, **snmp-vie.h**, and **snmpvars.c**.
3. The files `snmp-vie.h` and `snmpvars.c` are put in the SNMP folder.
4. The implementation for RFC1213 already exists in **rfc1213.c** in the SNMP folder.
5. The implementations for the rest of the MIBs are SNMPv3 dependent and are put in the file **v3mib.c** in the SNMPV3 folder.

The basic idea is as follows:

- **snmpvars.c** contains the list of all OIDs supported. All the MIBs are compiled with the "`-vh`" options, and the resulting **snmpvars.c** is placed in SNMP directory. The **.h** file **snmp_vie.h** contains declarations for all the prototypes in all the RFCs and is placed in the SNMP directory.
- RFC 1213's implementation is shipped with the SNMP agent so there is no point reimplementing it. To provide MIB instrumentation for the new RFCs, we compile the new RFCs (minus RFC 1213) with the -c

option to get a **.c** file with prototypes. This file is placed in the directory where the MIB instrumentation for new RFCs is to be provided, the SNMPV3 directory.

Suggested Data Structures

One way to optimize both speed and size in the SNMP agent is to use the suggested structures for holding data associated with MIB variables in groups and sequences. An example of how a large group can be implemented with minimal code is the implementation of the MIB-II ICMP group from the reference implementation. All the ICMP counters in this group are maintained in the suggested structure produced by the compiler in the **.h** file. The **C** routine only needs to use the "magic number" (also produced by the compiler) to index the ICMP structure as a table, and return the 32 bit quantity at the indicated offset. The code is reproduced below.

Magic numbers for the ICMP group and the suggested structure are produced automatically by the MIB compiler in the **.h** file:

```
/* tokens for 'icmp' group */
#define ICMPINMSGS              0
#define ICMPINERRORS            ICMPINMSGS+4
#define ICMPINDESTUNREACHS      ICMPINERRORS+4
#define ICMPINTIMEEXCDS         ICMPINDESTUNREACHS+4
#define ICMPINPARMPROBS         ICMPINTIMEEXCDS+4
#define ICMPINSRCQUENCHS        ICMPINPARMPROBS+4
#define ICMPINREDIRECTS         ICMPINSRCQUENCHS+4
#define ICMPINECHOS             ICMPINREDIRECTS+4
#define ICMPINECHOREPS          ICMPINECHOS+4
#define ICMPINTIMESTAMPS        ICMPINECHOREPS+4
#define ICMPINTIMESTAMPREPS     ICMPINTIMESTAMPS+4
#define ICMPINADDRMASKS         ICMPINTIMESTAMPREPS+4
#define ICMPINADDRMASKREPS      ICMPINADDRMASKS+4
#define ICMPOUTMSGS             ICMPINADDRMASKREPS+4
#define ICMPOUTERRORS           ICMPOUTMSGS+4
#define ICMPOUTDESTUNREACHS     ICMPOUTERRORS+4
#define ICMPOUTTIMEEXCDS        ICMPOUTDESTUNREACHS+4
#define ICMPOUTPARMPROBS        ICMPOUTTIMEEXCDS+4
#define ICMPOUTSRCQUENCHS       ICMPOUTPARMPROBS+4
#define ICMPOUTREDIRECTS        ICMPOUTSRCQUENCHS+4
#define ICMPOUTECHOS            ICMPOUTREDIRECTS+4
#define ICMPOUTECHOREPS         ICMPOUTECHOS+4
#define ICMPOUTTIMESTAMPS       ICMPOUTECHOREPS+4
#define ICMPOUTTIMESTAMPREPS    ICMPOUTTIMESTAMPS+4
#define ICMPOUTADDRMASKS        ICMPOUTTIMESTAMPREPS+4
#define ICMPOUTADDRMASKREPS     ICMPOUTADDRMASKS+4
```

The suggested structure is shown below:

```
 /* MIB table for 'icmp' group */

struct icmp_mib {
   u_long   icmpInMsgs;
   u_long   icmpInErrors;
   u_long   icmpInDestUnreachs;
   u_long   icmpInTimeExcds;
   u_long   icmpInParmProbs;
   u_long   icmpInSrcQuenchs;
   u_long   icmpInRedirects;
   u_long   icmpInEchos;
   u_long   icmpInEchoReps;
   u_long   icmpInTimestamps;
   u_long   icmpInTimestampReps;
   u_long   icmpInAddrMasks;
   u_long   icmpInAddrMaskReps;
   u_long   icmpOutMsgs;
   u_long   icmpOutErrors;
   u_long   icmpOutDestUnreachs;
   u_long   icmpOutTimeExcds;
   u_long   icmpOutParmProbs;
   u_long   icmpOutSrcQuenchs;
   u_long   icmpOutRedirects;
   u_long   icmpOutEchos;
   u_long   icmpOutEchoReps;
   u_long   icmpOutTimestamps;
   u_long   icmpOutTimestampReps;
   u_long   icmpOutAddrMasks;
   u_long   icmpOutAddrMaskReps;
};
```

The **var_icmp()** routine is based on a stub produced by the compiler in the **.c** file. It handles all 26 ICMP group variables and requires less than 10 lines of code to be added to the stub:

```
 u_char *
var_icmp(
   struct variable * vp,      /* IN - pointer to variables[ ] entry */
   oid *   name,              /* IN/OUT - input name requested; output name found */
   int *   length,            /* IN/OUT - length of input & output oids */
   int     oper,              /* IN - NEXT_OP (=0), GET_OP (=1), or SET_OP (=-1) */
   int *   var_len)           /* OUT - length of variable, or 0 if function */
{
u_char * cp;   /* return pointer */

   if(oper &&    /* GET or SET Object Ids must match exactly */
      (compare(name, *length, vp->name, (int)vp->namelen) != 0))
         return NULL;   /* return NULL if not exact match */

   /* The next two lines set the return variables. These are actually only needed
      for GETNEXTs - GETs and SETs already have an exact match. */
   memcpy(name, vp->name, (int)vp->namelen * sizeof(oid));
   *length = vp->namelen;
   *var_len = sizeof(long);   /* default length */

   cp = (u_char*)&icmp_mib;
   return(cp + vp->magic);
}
```

Of course it is not always practical to rewrite an existing system to use the suggested structures.

C Routine Frames

The C language routines referred to in the *variables[ ]* array are stubbed out further along in the **.c** file and prototyped in the **.h** file. The stub **var_system()** routine is shown below:

```
 u_char *
var_system(
   struct variable * vp,  /* IN - pointer to variables[] entry */
   oid *     name,        /* IN/OUT - input name requested; output name found */
   int *     length,      /* IN/OUT - length of input & output oids */
   int       oper,        /* IN - NEXT_OP (=0), GET_OP (=1), or SET_OP (=-1) */
   int *     var_len)     /* OUT - length of variable, or 0 if function */
{
   /* TODO: Add code here */
   return NULL;    /* default FAIL return. */
}
```

This routine is called by the SNMP agent code whenever an SNMP request is received with an `Object Id` that matches the routine's corresponding Object Id (name) in the variables table. Note that this stub produced by the compiler does no work and returns only a NULL. The meaning of the returned `NULL` varies depending on the setting of the Boolean `oper`. If exact is `TRUE`, then the SNMP datagram is a `SET` or `GET` command, and the returned `NULL` indicates that an exact match for the variable (passed in name) was not available. If exact is `FALSE`, then the request was a `GETNEXT` and the returned `NULL` means that no suitable GETNEXT Object Id was found by the routine.

If the routine had returned a non-`NULL` value, the return is a pointer to the variables data. The type of data pointed to is determined by the variable type in `vp->type`. In the non-`NULL` return case, the *name*, *`length`*, and *`var_len`* variables must be set to convey information about the data returned. The name is the `Object Id` of the returned variable. If the SNMP operation was a `SET` or `GET` (oper was non-zero) then this is the same as the name passed. If the operation was a `GETNEXT`, the porting programmer must update the name field. In indexed sequences this can be quite tricky! See the example for the "`At`" group in the files provided with the SNMP package. The length variable must be modified to reflect the length of the name returned. `var_len` must be set to the length in bytes, of the variable data returned. For 32 bit numeric returns such as `INTEGER`, `COUNT`, and `GAUGE` this will by 4. For Octet strings and Object Ids it will be the length of the string.

The filled in version of **var_system()** from **snmp/rfc1213.c** is shown below:

```
u_char *
var_system(
   struct variable *   vp,     /* IN - pointer to variables[ ] entry */
   oid *   name,               /* IN/OUT - input name requested; output name found */
   int *   length,             /* IN/OUT - length of input & output oids */
   int      oper,              /* IN - NEXT_OP (=0), GET_OP (=1), or SET_OP (=-1) */
   int *   var_len)            /* OUT - length of variable, or 0 if function */
{
   u_long   uptime;

   if(oper && (compare(name, *length, vp->name, (int)vp->namelen) != 0))
      return NULL;   /* GET or SET requires an exact match */

   /* The next two lines set the return variables. These are actually only needed
      for GETNEXTs - GETs and SETs already have an exact match. */
   memcpy(name, vp->name, (int)vp->namelen * sizeof(oid));
   *length = vp->namelen;
   *var_len = sizeof(long);   /* default length */

   switch (vp->magic)
   {
   case SYSDESCR:
      *var_len = strlen(sys_descr);
      return (u_char *)sys_descr;
   case SYSOBJECTID:
      *var_len = sizeof(sys_id);
      return (u_char *)sys_id;
   case SYSUPTIME:
      uptime = ((cticks * 100)/TPS);
      return (u_char *)&uptime;
   case SYSCONTACT:
      *var_len = strlen(sysContact);
      return (u_char *)sysContact;
   case SYSNAME:
      *var_len = strlen(sysName);
      return (u_char *)sysName;
   case SYSLOCATION:
      *var_len = strlen(sysLocation);
      return (u_char *)sysLocation;
   case SYSSERVICES:
      long_return = 0x0010L;
      return (u_char *)&long_return;
   default:
      SNMP_ERROR("var_system: Unknown magic number");
   }
   return NULL;   /* default FAIL return. */
}
```

4.4 Validation of Objects Defined in the MIB

The MIB Compiler can generate validation information for the objects read from the MIB. This information is used by the SNMP engine to validate objects at run time. This feature is optional and disabling it reduces the footprint of the SNMP engine. To enable/disable it, use the following line:

```
#define MIB_VALIDATION   1
```

When writing a MIB, SMI provides mechanisms to specify validation information for the MIB object.

```
 <enumerated values>
Integer32 (0..100)
Integer32 (0..100|300..500)
Integer32 (2|4|6|8)
OCTET STRING (SIZE(0..100))
OCTET STRING (SIZE(0..100|300..500))
OCTET STRING (SIZE(2|4|6|8))
```

When MIB_VALIDATION is enabled and the "-e" command line option is used, the MIB compiler parses this information and generates it in **snmpvars.c**. When this **snmpvars.c** file is used with the SNMP engine (which should also have MIB_VALIDATION enabled in **snmp_var.h**), validation of these objects is performed during the SEToperation. The MIB compiler also checks for all erroneous conditions (such as duplicate values, overlapping ranges, and so on) when parsing the validation information.

This table shows examples of illegal sub-typing (from RFC 1902):

Example code	Meaning
`Integer32 (150..100)`	First value greater than second.
`Integer32 (0..100|50..500)`	Two ranges overlap.
`Integer32 (0|2|0)`	Value is duplicated.
`Integer32 (MIN..-1|1..MAX)`	MIN and MAX not allowed.
`Integer32 (SIZE(0..34))`	Must not use SIZE.
`OCTET STRING (0..100)`	Must use SIZE.
`OCTET STRING (SIZE(-10..100))`	Negative SIZE.

The parsed information is stored in the *size_info* structure, as shown below:

```
#define MCT_MAX_VALUES 10

struct size_info {
   int itype ;      /* informationType - MCT_INT, MCT_STR, MCT_OTHER */
   int rtype ;      /* representationType - MCT_RANGES, MCT_VALUES */
   unsigned count ; /* Number of values/ranges */
   long values[MCT_MAX_VALUES];
};
```

The *values[ ]* field can store `MCT_MAX_VALUES` or `MCT_MAX_VALUES/2` ranges. When ranges are stored, *rtype* is `MCT_RANGES` and values appear in pairs in *values[ ]*. For example `(0..255|500..700)` will be `{0,255,500,700}`.

For enumerated values, this structure does not store the "text" for each value but simply stores the value. A separate structure of type *enum_list* is generated to store the complete information about the enumerated object. In this way, no information is lost and *size_info* stores sufficient information for validation.

`MCT_MAX_VALUES` defines the maximum number of values. If the MIB compiler finds that an object has more values than specified by `MCT_MAX_VALUES`, it generates a warning and ignores the subsequent values. The size of `MCT_MAX_VALUES` can be fine-tuned for this purpose.

If the porting engineer needs to do some special validation, they can use a special function that is called during the `SET`operation for the particular object. When such a function is defined for an object, the routine validation is not performed even if `MIB_VALIDATION` is enabled. This mechanism is provided so that the porting engineer can override the default method of validation.

To use the validation feature, `MIB_VALIDATION` must be enabled. In **snmp/snmp_var.h** enable `MIB_VALIDATION` and recompile the sources in this directory. This will enable the code in the SNMP engine which validates all incoming values of objects during the `SET` operation.

`MCT_MAX_VALUES` defines the maximum number of values that can be had, `10` by default. Maximum ranges that can be had are `(MCT_MAX_VALUES/2)`.

`MCT_MAX_VALUES` is also defined in **mibcomp/parse.h** and **snmp/snmp_var.h**. Hence if this value is changed, it must be changed in both places. The MIB compiler makes sure that it does not exceed this range. If more values have been specified in the MIB, a warning is generated and the subsequent values are discarded.

5 Application Programming Interface

This section documents the Application Programming Interface (API). It includes all the functions that are available to an application program.

5.1 Module Management

Whether you are using the Agent or Manager module (or both), you must run the base module. The functions are described in the following table.

Note: You must call any **register** functions after calling **snmp_base_init()** and before **snmp_base_start()**.

Function	Description
snmp_base_init()	Initializes the SNMP base module and allocates the required resources.
snmp_base_register_encryption()	Registers the 3DES encryption algorithm for use by an SNMP module.
snmp_base_register_hash()	Registers a hash algorithm used by an SNMP module.
snmp_register_file_ifc()	Registers a file interface callback descriptor for use by the SNMP Agent.
snmp_register_file_mib()	Registers the MIB interface for use by the SNMP Agent.
snmp_base_start()	Starts the SNMP base module.
snmp_base_stop()	Stops the SNMP base module.
snmp_base_delete()	Deletes the SNMP base module and releases the associated resources.

snmp_base_init

Use this function to initialize the SNMP base module and allocate the required resources.

Note: Call this before any other SNMP function.

Format

```
t_snmp_ret snmp_base_init ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

snmp_base_register_encryption

Use this function to register an encryption algorithm handle (obtained from the EEM) for use by the SNMP module.

Note: Call this after **snmp_base_init()** and before **snmp_base_start()**.

Format

```
t_snmp_ret snmp_base_register_encryption (
    t_snmp_encr        id,
    t_snmp_encr_par *   p_par )
```

Arguments

Argument	Description	Type
id	The encryption identifier. This is obtained from the Embedded Encryption Manager.	t_snmp_encr
p_par	A pointer to encryption parameters.	t_snmp_encr_par *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.

snmp_base_register_hash

Use this function to register a hash algorithm handle (obtained from the EEM) for use by the SNMP module.

> **Note:** Call this after **snmp_base_init()** and before **snmp_base_start()**.

Format

```
t_snmp_ret snmp_base_register_hash (
    t_snmp_auth     id,
    t_enc_ifc_hdl   hdl )
```

Arguments

Argument	Description	Type
id	The hash algorithm identifier.	t_snmp_auth
hdl	The handle of the hash algorithm. This is obtained from the Embedded Encryption Manager.	t_enc_ifc_hdl

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed: the handle was already registered.

snmp_register_file_ifc

Use this function to register a file interface callback descriptor for use by the SNMP Agent.

The file interface is used to store non-volatile data. Currently only one file, **snmp_usm.mib**, is used to store the boot count that is used by SNMPv3.

Note: Call this after **snmp_base_init()** and before **snmp_base_start()**.

Format

```
t_snmp_ret snmp_register_file_ifc ( const t_snmp_file_ifc_dsc * p_dsc )
```

Arguments

Argument	Description	Type
p_dsc	A pointer to the callback descriptor structure that describes the file interface.	t_snmp_file_ifc_dsc *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.

snmp_register_mib_ifc

Use this function to register a MIB interface for use by the SNMP Agent.

Note:

- Call this after **snmp_base_init()** and before **snmp_base_start()**.
- The OID number of the interface cannot be a subtree of an already registered MIB interface.

Format

```
t_snmp_ret snmp_register_mib_ifc ( const t_snmp_mib_ifc_dsc * const p_mib_dsc )
```

Arguments

Argument	Description	Type
p_mib_dsc	A pointer to the MIB descriptor structure.	t_snmp_mib_ifc_dsc *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed; there is no free space to register the MIB interface.
SNMP_PARAM_ERR	A parameter is invalid.
SNMP_ALREDY_REG_ERR	The interface is already registered.

snmp_base_start

Use this function to start the SNMP base module.

Note: Call **snmp_base_init()** and any **snmp_register_xxx()** functions before this function.

Format

```
t_snmp_ret snmp_base_start ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

snmp_base_stop

Use this function to stop the SNMP base module.

Format

```
t_snmp_ret snmp_base_stop ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

snmp_base_delete

Use this function to delete the SNMP base module and release the associated resources.

Format

```
t_snmp_ret snmp_base_delete ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

5.2 SNMP Manager

This section describes three types of function:

- Manager Functions - these are
- Utility Functions - these are used by the SNMP Manager, not by the Agent.
- Manager Callback Functions - there are two optional callbacks.

Manager Functions

The functions are as follows. For examples of these in use, see Code Examples.

Function	Description
snmp_connect()	Connects to an SNMP agent.
snmp_disconnect()	Disconnects a connection.
snmp_get_req()	Gets an OID from an SNMP agent, using a *GetRequest* command.
snmp_set_req()	Sets an OID object value in an SNMP agent, using a *SetRequest* command.
snmp_get_next_req()	Gets the next OID from an SNMP agent, using a *GetNextRequest* command.
snmp_get_bulk_req()	Gets MIB table data from an SNMP agent, using a *GetBulkRequest* command.
snmp_recv()	Gets a data buffer containing variable bindings from an SNMP agent.
snmp_release_buf()	Releases a buffer obtained by using **snmp_recv()**.

snmp_connect

Use this function to connect to an SNMP agent.

Format

```
t_snmp_ret snmp_connect (
    const t_snmp_conn_conf *   p_conf,
    t_snmp_ntf_cb              p_ntf_cb,
    t_snmp_conn_hdl *          p_hdl )
```

Arguments

Argument	Description	Type
p_conf	A pointer to a structure holding information on the configuration of the connection.	t_snmp_conn_conf *
p_ntf_cb	A callback function which can be called by the SNMP module to notify the user about connection events. This can be NULL.	t_snmp_ntf_cb
p_hdl	On return, a pointer to the SNMP connection handle.	t_snmp_conn_hdl *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_PARAM_ERR	A parameter is invalid.

Copyright HCC Embedded 2019

snmp_disconnect

Use this function to disconnect a connection.

The handle relates to an SNMP connection or a registered trap notify callback.

Format

```
t_snmp_ret snmp_disconnect( t_snmp_conn_hdl hdl )
```

Arguments

Argument	Description	Type
hdl	The handle of the connection or registered trap notify callback.	t_snmp_conn_hdl

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_PARAM_ERR	A parameter is invalid.

snmp_get_req

Use this function to get an OID from an SNMP agent.

This executes a *GetRequest* command. It sends an SNMP_PDU_GET_REQUEST PDU to the agent.

Format

```
t_snmp_ret snmp_get_req (
    t_snmp_conn_hdl   hdl,
    t_snmp_oid        oid_arr[],
    uint8_t           oid_nr )
```

Arguments

Argument	Description	Type
hdl	The connection handle.	t_snmp_conn_hdl
oid_arr[]	An array holding the Object IDs (OIDs) of variables that will be requested.	t_snmp_oid
oid_nr	The number of OIDs, the length of the *oid_arr[]* table.	uint8_t

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_PARAM_ERR	A parameter is invalid.

snmp_set_req

Use this function to set an OID object value in an SNMP agent.

This executes a *SetRequest* command. It sends an SNMP_PDU_SET_REQUEST PDU to the agent.

Format

```
t_snmp_ret snmp_set_req (
    t_snmp_conn_hdl   hdl,
    t_snmp_oid        oid_arr[],
    t_snmp_mib_obj    val_arr[],
    uint8_t           oid_nr )
```

Arguments

Argument	Description	Type
hdl	The connection handle.	t_snmp_conn_hdl
oid_arr[]	An array holding the Object ID (OID) data.	t_snmp_oid
val_arr[]	An array holding the OID values.	t_snmp_mib_obj
oid_nr	The number of OIDs in the *oid_arr[]* array.	uint8_t

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_PARAM_ERR	A parameter is invalid.

snmp_get_next_req

Use this function to get the next OID from an SNMP agent.

This executes a *GetNextRequest* command. It sends an SNMP_PDU_GET_NEXT_REQUEST PDU to the agent.

Format

```
t_snmp_ret snmp_get_next_req (
    t_snmp_conn_hdl    hdl,
    t_snmp_oid         oid_arr[],
    uint8_t            oid_nr )
```

Arguments

Argument	Description	Type
hdl	The connection handle.	t_snmp_conn_hdl
oid_arr[]	An array holding the Object IDs (OIDs) of variables that will be requested.	t_snmp_oid
oid_nr	The number of OIDs, the length of the *oid_arr[]* array.	uint8_t

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_PARAM_ERR	A parameter is invalid.

snmp_get_bulk_req

Use this function to get MIB table data from an SNMP agent.

This executes a *GetBulkRequest* command. It sends an SNMP_PDU_GET_BULK_REQUEST PDU to the agent to retrieve data from the MIB.

The *non_repeaters* and *max_repetitions* fields are used to control response behavior.

Format

```
t_snmp_ret snmp_get_bulk_req (
    t_snmp_conn_hdl   hdl,
    t_snmp_oid        oid_arr[],
    uint8_t           oid_nr,
    uint8_t           non_repeaters,
    uint8_t           max_repetitions )
```

Arguments

Argument	Description	Type
hdl	The connection handle.	t_snmp_conn_hdl
oid_arr[]	An array holding the Object IDs (OIDs) of variables that will be requested.	t_snmp_oid
oid_nr	The length of the *oid_arr[]* array.	uint8_t
non_repeaters	The number of non-repeating OIDs in the *oid_arr[]* table. These are the first elements in the table.	uint8_t
max_repetitions	The number of variables requested for repeating OIDs.	uint8_t

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_PARAM_ERR	A parameter is invalid.

snmp_recv

Use this function to get a data buffer containing variable bindings from the SNMP agent.

Note: After this call, release the buffer it obtains by calling **snmp_release_buf()**.

Format

```
t_snmp_ret snmp_recv (
    t_snmp_conn_hdl    hdl,
    t_uint8_t * *      pp_msg,
    uint16_t *         p_msg_len,
    uint32_t *         p_status
    uint32_t *         p_err_index )
```

Arguments

Argument	Description	Type
hdl	The connection handle.	t_snmp_conn_hdl
pp_msg	On return, a pointer to the data buffer. Data in the buffer is a list of variable bindings in X.690 format.	t_uint8_t * *
p_msg_len	On return, a pointer to the number of received bytes. This can be NULL.	uint16_t *
p_status	On return, a pointer to the variable that received the PDU message status. This can be NULL.	uint32_t *
p_err_index	On return, a pointer to the variable that received the PDU message error index. This can be NULL.	uint32_t *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_PARAM_ERR	A parameter is invalid.

snmp_release_buf

Use this function to release a buffer obtained by using **snmp_recv()**.

Format

```
void snmp_release_buf ( uint8_t * p_msg )
```

Arguments

Argument	Description	Type
p_msg	A pointer to the buffer.	uint8_t *

Return Values

None.

Utility Functions

These functions are used by the SNMP Manager, not the Agent. They are described in the following table.

Function	Description
snmp_get_varbind()	Parses an SNMP message obtained by **snmp_recv()** and extracts the Object Identifier (OID) and its value.
snmp_obj_rd_int32()	Converts a signed integer 32 bit number in Basic Encoding Rules (BER) format for use by the SNMP Manager.
snmp_obj_rd_int64()	Converts a signed integer 64 bit number in Basic Encoding Rules (BER) format for use by the SNMP Manager.
snmp_obj_rd_uint32()	Converts an unsigned integer 32 bit number in BER format for use by the SNMP Manager.
snmp_obj_wr_int32()	Converts a signed integer 32 bit value into BER format and writes it.
snmp_obj_wr_int64()	Converts a signed integer 64 bit value into BER format and writes it.
snmp_obj_wr_uint32()	Converts an unsigned integer value into BER format and writes it.
snmp_write_oid()	Converts an unsigned integer array OID number into BER format for use by the SNMP Manager.
snmp_write_oid_obj()	Converts an unsigned integer array OID number into BER format for use by the SNMP Manager.
snmp_get_snmp_mib2_stats()	Gets SNMP statistics.

snmp_get_varbind

Use this function to parse an SNMP message obtained by using **snmp_recv()** and extract the OID and its value.

Format

```
t_snmp_ret snmp_get_varbind(
    uint8_t             p_msg[],
    uint16_t            msg_len,
    t_snmp_oid *        p_oid,
    t_snmp_mib_obj *    p_obj,
    uint16_t *          p_var_len )
```

Arguments

Argument	Description	Type
p_msg[]	A pointer to the message buffer.	uint8_t
msg_len	The message buffer length.	uint16_t
p_oid	On return, a pointer to the OID structure.	t_snmp_oid *
p_obj	On return, a pointer to the object structure containing the integer in BER format.	t_snmp_mib_obj *
p_var_len	On return, a pointer to the current variable binding length (this can be NULL).	uint16_t *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_ASN_SYNT_ERR	ASN syntax error in received PDU.
SNMP_PDU_SYNT_ERR	Message syntax error.
SNMP_PARAM_ERR	Invalid input parameter.

snmp_obj_rd_int32

Use this function to convert a signed integer 32 bit number value in BER format to *int32_t* format.

Format

```
t_snmp_ret snmp_obj_rd_int32 (
    t_snmp_mib_obj *   p_obj,
    int32_t *          p_val )
```

Arguments

Argument	Description	Type
p_obj	A pointer to the object structure containing the integer in BER format.	t_snmp_mib_obj *
p_val	A pointer to the variable that will receive the converted value.	int32_t *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_PARAM_ERR	Invalid input parameter.

snmp_obj_rd_int64

Use this function to convert a signed integer 64 bit number value in BER format to *int64_t* format.

Format

```
t_snmp_ret snmp_obj_rd_int64 (
    t_snmp_mib_obj *   p_obj,
    int64_t *          p_val )
```

Arguments

Argument	Description	Type
p_obj	A pointer to the object structure containing the integer in BER format. This must be a valid structure.	t_snmp_mib_obj *
p_val	A pointer to the variable that will receive the converted value.	int64_t *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_PARAM_ERR	Invalid input parameter.

snmp_obj_rd_uint32

Use this function to convert an unsigned 32 bit number in BER format to *uint32_t* format.

Format

```
t_snmp_ret snmp_obj_rd_int32 (
    t_snmp_mib_obj *   p_obj,
    uint32_t *         p_val )
```

Arguments

Argument	Description	Type
p_obj	A pointer to the object containing the integer in BER format.	t_snmp_mib_obj *
p_val	A pointer to the variable that will receive the converted value.	uint32_t *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_PARAM_ERR	Invalid input parameter.

Copyright HCC Embedded 2019

snmp_obj_wr_int32

Use this function to convert a signed integer value into BER format and write it.

The envelope type is not changed by this function.

Format

```
t_snmp_ret snmp_obj_wr_int32 (
    int32_t             val,
    t_snmp_mib_obj *    p_obj )
```

Arguments

Argument	Description	Type
val	The value to be written.	int32_t
p_obj	A pointer to the variable that will receive the converted value.	t_snmp_mib_obj *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_PARAM_ERR	Invalid input parameter.

snmp_obj_wr_int64

Use this function to convert a signed integer value into BER format and write it.

The envelope type is not changed by this function.

Note: *p_obj->snmo_len* must be set to a length of *p_obj->p_snmo_obuf*. The size of *p_obj>p_snmo_obuf* must be at least 9.

Format

```
t_snmp_ret snmp_obj_wr_int64 (
    int64_t            val,
    t_snmp_mib_obj *   p_obj )
```

Arguments

Argument	Description	Type
val	The value to be written.	int64_t
p_obj	A pointer to the variable that will receive the converted value.	t_snmp_mib_obj *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_PARAM_ERR	Invalid input parameter.

Copyright HCC Embedded 2019

snmp_obj_wr_uint32

Use this function to convert an unsigned integer value into BER format and write it.

The envelope type is not changed by this function.

Format

```
t_snmp_ret snmp_obj_wr_uint32 (
    uint32_t          val,
    t_snmp_mib_obj *   p_obj )
```

Arguments

Argument	Description	Type
val	The value to be written.	uint32_t
p_obj	A pointer to the variable that will receive the converted value.	t_snmp_mib_obj *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_PARAM_ERR	Invalid input parameter.

snmp_write_oid

Use this function to convert an Object Identifier (OID) into BER format for use by the SNMP Manager.

Format

```
t_snmp_ret snmp_write_oid(
    uint32_t       p_oid_arr[],
    uint8_t        oid_length,
    t_snmp_oid *   p_oid );
```

Arguments

Argument	Description	Type
p_oid_arr[]	A pointer to the array containing the OID value.	uint32_t
oid_length	The message buffer length.	uint8_t
p_oid	A pointer to the output OID structure.	t_snmp_oid *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_BUFF_ERR	The buffer provided for writing the OID is too small.
SNMP_PARAM_ERR	Invalid input parameter.

snmp_write_oid_obj

Use this function to convert an OID number into BER format for use by the SNMP Manager.

Format

```
t_snmp_ret snmp_write_oid_obj (
    uint32_t          p_oid_arr[],
    uint8_t           oid_length,
    t_snmp_mib_obj *   p_obj );
```

Arguments

Argument	Description	Type
p_oid_arr[]	A pointer to the array containing the OID value.	uint32_t
oid_length	The message buffer length.	uint8_t
p_obj	A pointer to the object that will contain the OID in BER format.	t_snmp_mib_obj *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution; PDU parsed.
SNMP_BUFF_ERR	The buffer provided for writing the OID is too small.
SNMP_PARAM_ERR	Invalid input parameter.

snmp_get_snmp_mib2_stats

Use this function to get SNMP statistics.

Note: This only applies if configuration option SNMP_MIB2_SNMP_ENABLE is set.

Format

```
t_snmp_ret snmp_get_snmp_mib2_stats ( t_mib2_snmp * p_mib )
```

Arguments

Argument	Description	Type
p_mib	Where to write the MIB statistics.	t_mib2_snmp *

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_PARAM_ERR	Invalid input parameter.

Manager Callback Functions

The callback functions are listed in the following table.

Note: It is the user's responsibility to provide any callback functions the application requires. Providing such functions is optional.

Function	Description
t_snmp_conn_cb()	Specifies the format of the callback function that you can use to notify a user about connection events.
t_snmp_ntf_cb()	Specifies the format of the callback function that you can use to notify a user about an incoming notification.

t_snmp_conn_cb

The **t_snmp_conn_cb()** definition specifies the format of the callback function that you can use to notify a user about connection events.

Note: It is the user's responsibility to provide this callback and its use is optional.

Format

```
typedef void ( * t_snmp_conn_cb ) ( uint32_t ntf )
```

Arguments

Argument	Description	Type
ntf	The notification function.	uint32_t

Copyright HCC Embedded 2019

t_snmp_ntf_cb

The **t_snmp_ntf_cb()** definition specifies the format of the callback function which you can use to notify a user about an incoming notification.

Note: It is the user's responsibility to provide this callback and its use is optional.

Format

```
typedef void ( * t_snmp_ntf_cb ) ( t_snmp_ntf ntf )
```

Arguments

Argument	Description	Type
ntf	The notification function.	t_snmp_ntf

5.3 SNMP Agent

This section describes three types of function:

- File Interface Functions - these provide the interface to the SNMP server.
- Agent Functions - these are used to send a PDU or register a callback.
- Agent Callback Functions - there are five optional callbacks.

File Interface Functions

These functions are the interface to the SNMP server. You can implement these as required.

Note: It is the user's responsibility to provide any callback functions the application requires. Providing such functions is optional.

The functions are the following:

Function	Description
t_snmp_get()	Specifies the format of the function that you can use to obtain data from the MIB.
t_snmp_get_next()	Specifies the format of the function that you can use to obtain the next object from the MIB.
t_snmp_set()	Specifies the format of the function that you can use to set the value of a MIB object.

t_snmp_get

The **t_snmp_get()** definition specifies the format of the function that you can use to obtain data from the MIB.

Format

```
typedef t_snmp_ret ( * t_snmp_get ) (
    const uint8_t *       p_contextName,
    uint8_t               context_len,
    const t_snmp_oid *    p_oid,
    t_snmp_mib_obj *      p_obj )
```

Arguments

Argument	Description	Type
p_contextName	A pointer to the name of the SNMP context.	uint8_t *
context_len	The length of the context name.	uint8_t
p_oid	A pointer to the object ID of the data to be obtained.	t_snmp_oid *
p_obj	On return, a pointer to the object.	t_snmp_mib_obj *

Return Values

Code	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

t_snmp_get_next

The **t_snmp_get_next()** definition specifies the format of the function that you can use to obtain the next object from the MIB. (This is the object after that obtained by using **t_snmp_get()**.)

Format

```
typedef t_snmp_ret ( * t_snmp_get_next ) (
    const uint8_t *    p_contextName,
    uint8_t            context_len,
    t_snmp_oid *       p_oid,
    t_snmp_mib_obj *   p_obj )
```

Arguments

Argument	Description	Type
p_contextName	A pointer to the name of the SNMP context.	uint8_t *
context_len	The length of the context name.	uint8_t
p_oid	A pointer to the object ID.	t_snmp_oid *
p_obj	On return, a pointer to the object.	t_snmp_mib_obj *

Return Values

Code	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

t_snmp_set

The **t_snmp_set()** definition specifies the format of the function that you can use to set the value of a MIB object.

Format

```
typedef t_snmp_ret ( * t_snmp_set ) (
    const uint8_t *          p_contextName,
    uint8_t                  context_len,
    const t_snmp_oid *       p_oid,
    const t_snmp_mib_obj *   p_obj )
```

Arguments

Argument	Description	Type
p_contextName	A pointer to the name of the SNMP context.	uint8_t *
context_len	The length of the context name.	uint8_t
p_oid	A pointer to the object ID.	t_snmp_oid *
p_obj	On return, a pointer to the object.	t_snmp_mib_obj *

Return Values

Code	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

Agent Functions

The functions are the following:

Function	Description
snmp_send_ntf()	Sends a notification PDU (a "trap").
snmp_reg_cb()	Registers a callback function for the SNMP agent. The callback is used by the agent to notify users of the result of any traps it sends.

snmp_send_ntf

Use this function to send a notification PDU (a "trap") to an SNMP Manager.

The type of notification and targets are defined by a notify filtering mechanism.

Format

```
t_snmp_ret snmp_send_ntf (
    const t_snmp_oid *   p_oid,
    uint8_t *            p_msg,
    uint16_t             msg_len )
```

Arguments

Argument	Description	Type
p_oid	A pointer to the structure containing the trap OID.	t_snmp_oid *
p_msg	A pointer to the buffer holding the variable binding list that can be added to the trap message.	uint8_t *
msg_len	The length of *p_msg*.	uint16_t

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.
SNMP_PARAM_ERR	Invalid parameter.

snmp_reg_cb

Use this function to register a callback function for the SNMP agent. The callback is used by the agent to notify users of the result of any traps it sends.

Note: Call this after **snmp_base_init()** and before **snmp_base_start()**.

Format

```
t_snmp_ret snmp_reg_cb ( t_snmp_ntf_cb  p_cb )
```

Arguments

Argument	Description	Type
p_cb	A pointer to the SNMP callback function.	t_snmp_ntf_cb

Return Values

Return value	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

Agent Callback Functions

The callback functions are listed in the following table.

Note: It is the user's responsibility to provide any callback functions the application requires. Providing such functions is optional.

Function	Description
t_snmp_open_cb()	Specifies the format of the function that you can use to open a file.
t_snmp_close_cb()	Specifies the format of the function that you can use to close a file.
t_snmp_read_cb()	Specifies the format of the function that you can use to read from a file.
t_snmp_write_cb()	Specifies the format of the function that you can use to write to a file.
t_snmp_seek_cb()	Specifies the format of the function that you can use to set the position indicator within a file.

t_snmp_open_cb

The **t_snmp_open_cb()** definition specifies the format of the callback function which you can use to open a file.

Format

```
typedef t_snmp_ret ( * t_snmp_open_cb ) (
    char_t * const          p_filename,
    const t_snmp_file_mode  mode,
    uint32_t * const        p_hdl )
```

Arguments

Argument	Description	Type
p_filename	A pointer to the file name.	char_t *
mode	The mode to open the file in.	t_snmp_file_mode
p_hdl	On return, a pointer to the file handle.	uint32_t *

Return Codes

Code	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

t_snmp_close_cb

The **t_snmp_close_cb()** definition specifies the format of the callback function which you can use to close a file.

Format

```
typedef void ( * t_snmp_close_cb ) ( const uint32_t hdl )
```

Arguments

Argument	Description	Type
hdl	The file handle.	uint32_t

Return Codes

Code	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

t_snmp_read_cb

The **t_snmp_read_cb()** definition specifies the format of the callback function which you can use to read from a file.

Format

```
typedef t_snmp_ret ( * t_snmp_read_cb ) (
    const uint32_t     hdl,
    uint8_t * const    p_buf,
    const uint16_t     buf_len,
    uint16_t * const   p_rd_len )
```

Arguments

Argument	Description	Type
hdl	The file handle.	uint32_t
p_buf	A pointer to the buffer to read into.	uint8_t *
buf_len	The buffer length in bytes.	uint16_t
p_rd_len	On return, the number of bytes read.	uint16_t *

Return Codes

Code	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

t_snmp_write_cb

The **t_snmp_write_cb()** definition specifies the format of the callback function which you can use to write to a file.

Format

```
typedef t_snmp_ret ( * t_snmp_write_cb ) (
    const uint32_t      hdl,
    uint8_t * const     p_buf,
    const uint16_t      buf_len,
    uint16_t * const    p_wr_len )
```

Arguments

Argument	Description	Type
hdl	The file handle.	uint32_t
p_buf	A pointer to the buffer to write to.	uint8_t *
buf_len	The buffer length in bytes.	uint16_t
p_wr_len	On return, the number of bytes written.	uint16_t *

Return Codes

Code	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

t_snmp_seek_cb

The **t_snmp_seek_cb()** definition specifies the format of the callback function which you can use to set the position indicator within a file.

Format

```
typedef t_snmp_ret ( * t_snmp_seek_cb ) (
    const uint32_t   hdl,
    uint32_t         pos )
```

Arguments

Argument	Description	Type
hdl	The file handle.	uint32_t
pos	The position within the file.	uint32_t

Return Codes

Code	Description
SNMP_SUCCESS	Successful execution.
SNMP_ERROR	Operation failed.

5.4 Error Codes

If a function executes successfully, it returns with SNMP_SUCCESS, a value of 0. The following table shows the meaning of the SNMP error codes.

Return Value	Value	Description
SNMP_SUCCESS	0	Successful execution.
SNMP_ERROR	1	General error.
SNMP_ASN_SYNT_ERR	2	The message ASN syntax is incorrect.
SNMP_AUTH_ERR	3	Authentication error.
SNMP_BUFF_ERR	4	There is not enough space in the send buffer.
SNMP_NO_SUCH_OBJECT	5	There is no such object in the MIB.
SNMP_NO_SUCH_INST	6	There is no such instance in the MIB.
SNMP_END_OF_MIB	7	Requesting an object from MIB reached end of MIB view.
SNMP_MIB_NO_ACCESS	8	The user does not have access rights to the requested object.
SNMP_MIB_WRONG_TYPE	9	The object type does not match MIB specification object type.
SNMP_MIB_WRONG_LENGTH	10	The MIB object to be set has the wrong length.
SNMP_PDU_SYNT_ERR	14	The message syntax is incorrect.
SNMP_PARAM_ERR	15	A parameter is invalid.
SNMP_WRONG_ENG_ID	16	Wrong engine ID in message.
SNMP_WRONG_UN	17	Wrong user name in message.
SNMP_DROP_MSG	18	Drop current message.
SNMP_SECLEV_ERR	19	Security level error.
SNMP_TIME_ERR	20	Time window error.
SNMP_NO_PDUH_ERR	21	No handler for this PDU exists.
SNMP_ALREDY_REG_ERR	22	The interface is already registered.

Note: Also check error code values in the base system by using the HCC TCP/IP Dual Stack System User Guide.

5.5 Error Status Values

The following table shows the meaning of the SNMP error status values.

Return Value	Value	Description
SNMP_ERR_STS_NOERR	0	No error.
SNMP_ERR_STS_TOOBIG	1	Packet too big.
SNMP_ERR_STS_NOSUCHNAME	2	No such name.
SNMP_ERR_STS_BADVALUE	3	Bad value.
SNMP_ERR_STS_READONLY	4	Read-only.
SNMP_ERR_STS_GENERR	5	General error.
SNMP_ERR_STS_NOACCESS	6	No access.
SNMP_ERR_STS_WRTYPE	7	Wrong type.
SNMP_ERR_STS_WRLENGTH	8	Wrong length.
SNMP_ERR_STS_WRENCODING	9	Wrong coding.
SNMP_ERR_STS_WRVALUE	10	Wrong value.
SNMP_ERR_STS_NOCREATION	11	No creation.
SNMP_ERR_STS_INCON_VALUE	12	Inconsistent value.
SNMP_ERR_STS_RES_UNAVAILABLE	13	Resource unavailable.
SNMP_ERR_STS_COMMIT_FAILED	14	Commit failed.
SNMP_ERR_STS_UNDO_FAILED	15	Undo failed.
SNMP_ERR_STS_AUTH_ERR	16	Authorization error.
SNMP_ERR_STS_NOTWRITABLE	17	Not writable.
SNMP_ERR_STS_INCON_NAME	18	Inconsistent name.

5.6 MIB Status Values

The following table shows the meaning of the MIB status values.

Return Value	Description
SNMP_MIB_NOERR	No error.
SNMP_MIB_TOOBIG	Packet too big.
SNMP_MIB_NOSUCHNAME	No such name.
SNMP_MIB_BADVALUE	Bad value.
SNMP_MIB_READONLY	Read-only.
SNMP_MIB_GENERR	General error.
SNMP_MIB_NOACCESS	No access.
SNMP_MIB_WRTYPE	Wrong type.
SNMP_MIB_WRLENGTH	Wrong length.
SNMP_MIB_WRENCODING	Wrong coding.
SNMP_MIB_WRVALUE	Wrong value.
SNMP_MIB_NOCREATION	No creation.
SNMP_MIB_INCON_VALUE	Inconsistent value.
SNMP_MIB_RES_UNAVAILABLE	Resource unavailable.
SNMP_MIB_COMMIT_FAILED	Commit failed.
SNMP_MIB_UNDO_FAILED	Undo failed.
SNMP_MIB_AUTH_ERR	Authorization error.
SNMP_MIB_NOTWRITABLE	Not writable.
SNMP_MIB_INCON_NAME	Inconsistent name.

5.7 Types and Definitions

PDU Types

This table describes the types of PDU that are handled by SNMP:

PDU Name	Value	Description
SNMP_PDU_GET_REQUEST	0	*GetRequest* PDU sent to an agent by the SNMP manager.
SNMP_PDU_GET_NEXT_REQUEST	1	*GetNextRequest* PDU sent to an agent by the SNMP manager.
SNMP_PDU_RESPONSE_PDU	2	Response message PDU sent to the SNMP manager by an SNMP agent.
SNMP_PDU_SET_REQUEST	3	Set object value message PDU sent by the SNMP manager to an agent to update the value of an SNMP object.
SNMP_PDU_GET_BULK_REQUEST	5	*GetBulkRequest* PDU sent by an agent to retrieve data from MIB tables.
SNMP_PDU_INFORM_REQUEST	6	Information PDU sent to the SNMP manager by an agent to notify of an event. This message must be acknowledged by sending a response PDU.
SNMP_PDU_TRAP	7	Trap PDU sent to the SNMP manager by an agent to notify of an event. This message does not need to be acknowledged.
SNMP_PDU_REPORT	8	Report PDU sent to the SNMP manager by an agent to notify of an error during processing of a PDU.

Connection Events

The following events may be reported during connection:

Event Name	Value	Description
SNMP_CONN_EST	0x01	Connection established.
SNMP_CONN_TX	0x02	Data transmitted.
SNMP_CONN_RX	0x04	Data received.
SNMP_CONN_TIMEOUT	0x08	Command replay timeout.
SNMP_CONN_ERR	0x10	Error during connection.
SNMP_CONN_NTF_RPL	0x20	Successfully replied to notification PDU.

t_snmp_mib_obj

The *t_snmp_mib_obj* structure describes an SNMP MIB value object. Its components are as follows:

Element	Type	Description
p_snmo_obuf	uint8_t *	A pointer to object data.
snmo_len	uint16_t	The object length.
snmo_ber_env	uint8_t	The BER object envelope.

t_snmp_oid

The *t_snmp_oid* structure describes an SNMP Object Identifier (OID). Its components are as follows:

Element	Type	Description
spo_length	uint8_t	The OID length.
p_spo_oid	uint8_t *	A pointer to the buffer containing the OID value in X. 690 notation.

t_snmp_encr_par

The *t_snmp_encr_par* structure describes encryption driver parameters:

Element	Type	Description
snep_hdl	t_enc_ifc_hdl	The Embedded Encryption Manager handle of the encryption driver.
snep_key_length	uint16_t	The key length.
snep_iv_length	uint16_t	The initialization vector length.
snep_block_size	uint16_t	The encryption block size. Set this to 0 if encryption does not require the input data size to be multiple of block size.

t_mib2_snmp

The *t_mib2_snmp* structure describes MIB specifications. Its components are as follows:

Element	Type	Description
mibsn_in_pkts	uint32_t	The input packets counter.
mibsn_out_pkts	uint32_t	The output packets counter.
mibsn_silent_drops	uint32_t	The silent drop packets counter.

t_snmp_file_mode

A file can be opened in the following modes, defined by *t_snmp_file_mode*:

Element	Description
SMNP_MODE_READ	Read mode.
SNMP_MODE_WRITE	Write mode.

t_snmp_file_ifc_dsc

The **t_snmp_file_ifc_dsc** structure defines the file system interface callbacks. Its components are as follows:

Element	Type	Description
fcd_open_cb	t_snmp_open_cb	The open callback.
fcd_close_cb	t_snmp_close_cb	The close callback.
fcd_read_cb	t_snmp_read_cb	The read callback.
fcd_write_cb	t_snmp_write_cb	The write callback.
fcd_seek_cb	t_snmp_seek_cb	The seek callback.

t_snmp_mib_ifc_dsc

The *t_snmp_mib_ifc_dsc* typedef defines the MIB interface. The module uses this structure to define the interface for communicating with a user MIB.

Its components are as follows:

Element	Type	Description
mibi_oid_subtree	t_snmp_oid	The MIB subtree Object Identifier (OID). OID length must be less than SNMP_MAX_OID_LENGTH.
mibi_get_cb	t_snmp_get	The get MIB object callback.
mibi_get_next_cb	t_snmp_get_next	The get next MIB object callback.
mibi_set_cb	t_snmp_set	The set MIB object value callback.

Copyright HCC Embedded 2019

t_snmp_auth and t_snmp_encr

A keyed-Hash Message Authentication Code (HMAC) uses a combination of a hash algorithm and an encryption key to produce a Message Authentication Code (MAC). Depending on the algorithm used, the MAC algorithm is referred to as HMAC-MD5-96 or HMAC-SHA-96.

The types *t_snmp_auth* and *t_snmp_encr* are used to produce the MAC.

t_snmp_auth

Type *t_snmp_auth* is used to determine the hash algorithm.

Element	Description
SNMP_AUTH_NONE	Do not use authorization.
SNMP_AUTH_HMAC_MD5_96	Use the Message Digest 5 (MD5) algorithm.
SNMP_AUTH_HMAC_SHA_96	Use the Secure Hash Algorithm (SHA).

t_snmp_encr

Type *t_snmp_encr* is used to determine the encryption algorithm.

Element	Description
SNMP_ENCR_NONE	Do not use an encryption algorithm.
SNMP_ENCR_DES	Use the Triple Data Encryption Standard (3DES).
SNMP_ENCR_AES128	Use the Advanced Encryption Standard (AES) 128.
SNMP_ENCR_AES256	Use the Advanced Encryption Standard (AES) 256.

t_snmp_community

The typedef *t_snmp_community* is used to describe a community. Its components are as follows:

Element	Type	Description
p_snc_name	uint8_t *	The community name (a character string).
snc_name_len	uint8_t	The length of the community name.
snc_addr	uint32_t	The address of the allowed peer.
snc_addr_mask	uint32_t	The allowed peer's address mask.
snu_rights	uint8_t	The community's access rights.

Access Rights

The access rights used by *t_snmp_community* and *t_snmp_user* are as follows:

Mode	Value	Description
SNMP_COMM_ACCESS_RO	0	Read only.
SNMP_COMM_ACCESS_RW	1	Read/write.

t_snmp_conn_conf

The typedef *t_snmp_conn_conf* describes parameters needed during manager connection. Its components are as follows:

Element	Type	Description
sncc_port	t_ip_port	The port number.
sncc_version	uint8_t	The SNMP protocol version.
sncc_flags	uint8_t	Connection flags (SNMP_CFLAGS_KEY).
p_sncc_user	uint8_t *	The user name.
sncc_user_len	uint8_t	The length of the user name.
p_sncc_auth_pass	uint8_t *	The authorization password.
sncc_auth_len	uint8_t	The length of the authorization password.
sncc_auth	t_snmp_auth	The authorization mode.
p_sncc_encr_pass	uint8_t *	The encryption password.
sncc_encr_len	uint8_t	The length of the encryption password.
sncc_encr	t_snmp_encr	The encryption mode.
p_sncc_cont_name	uint8_t *	The context name.
sncc_ cenm_len	uint8_t	The length of the community/context name.
sncc_recon	uint8_t	The number of reconnection attempts.
sncc_timeout	uint16_t	The reply timeout in 0.01 seconds.

Copyright HCC Embedded 2019

t_snmp_user

The typedef *t_snmp_user* is used to describe users when SNMPv3 is enabled. Its components are as follows:

Element	Type	Description
p_snu_name	uint8_t *	The user name (a character string).
snu_name_len	uint8_t	The length of the user name.
p_snu_pass	uint8_t *	The user password (a character string).
snu_pass_len	uint8_t	The length of the user password.
p_snu_epass	uint8_t *	The encryption password (a character string).
snu_epass_len	uint8_t	The length of the encryption password.
snu_auth	t_snmp_auth	The authorization mode.
snu_encr	t_snmp_encr	The encryption mode.
snu_flags	uint8_t	The user's access rights (SNMP_TFLAGS_ACCESS_*).

t_snmp_ntf

The typedef *t_snmp_ntf* is used to describe notifications. Its components are as follows:

Element	Type	Description
snn_version	uint8_t	The protocol version.
p_snn_name	uint8_t *	The user or community name. This depends on the protocol version.
snn_name_len	uint8_t	The length of the *p_snn_name* buffer.
snn_user_id	uint8_t	The index number of a user entry in the *g_snmp_users* table that is assigned to the notification.
snn_ntf_type	uint8_t	The notification PDU type.
snn_recon	uint32_t	The number of reconnection attempts.
snn_timeout	uint32_t	The reply timeout in 0.01 second units.

Notification Types

The following notifications may be reported:

Element	Type	Description
SNMP_NTF_TYPE_TRAP	1	Notification PDU of type trap.
SNMP_NTF_TYPE_NOTIFICATION	2	Notification PDU of type notify.

The following notifications may be sent by the agent's **snmp_reg_cb()** function:

Element	Value	Description
SNMP_NTF_CONF_ERR	1	Did not receive confirmation for trap sent.
SNMP_CONN_NTF_RPL	2	Send all traps (if traps were not confirmed, this event is sent when message will be exceed resend number).

t_snmp_trap

The *t_snmp_trap* typedef defines a trap. Its components are as follows:

Element	Type	Description
snn_version	uint8_t	The protocol version.
p_snn_name	uint8_t *	The context or community name, depending on the protocol version.
snn_name_len	uint8_t	The length of the *p_snn_name* buffer.
snn_user_id	uint8_t	The index number of user entry in *g_snmp_users[]* table that is assigned to the notify.
snn_ntf_type	uint8_t	The notification PDU type.
snn_recon	uint32_t	The number of reconnection
snn_timeout	uint16_t	The reply timeout in 0.01 second units.
snn_addr	t_ip_port	The notify target address.

SNMP Manager Connection States

The following table shows the meaning of the SNMP manager connection states.

Return Value	Value	Description
SNMP_CONN_TRAP	1	Received trap.
SNMP_CONN_EST	2	Connection established.
SNMP_CONN_TX	4	Message transmitted.
SNMP_NTF_CONF_ERR	8	Notify confirmation error.
SNMP_CONN_RX	0x10	Received message.
SNMP_CONN_TIMEOUT	0x20	Connection timeout.
SNMP_CONN_ERR	0x40	Connection error.
SNMP_CONN_NTF_RPL	0x80	Notify was replied.

SNMP Version Numbers

The following table shows the SNMP version numbers.

Return Value	Value	Description
SNMP_VER_SNMP1	0	SNMPv1.
SNMP_VER_SNMP2VC	1	SNMPv2c.
SNMP_VER_SNMP3	3	SNMPv3.

Object Types

The following object types are used by the SNMP Manager:

Element	Type	Description
SNMP_MAN_OBJECT_INTEGER	0x02	Integer.
SNMP_MAN_OBJECT_BITS	0x03	Bits.
SNMP_MAN_OBJECT_STRING	0x04	String.
SNMP_MAN_OBJECT_NULL	0x05	NULL.
SNMP_MAN_OBJECT_OID	0x06	Object identifier.
SNMP_MAN_OBJECT_IP_ADDR	0x40	IP address.
SNMP_MAN_OBJECT_COUNTER	0x41	Unsigned integer.
SNMP_MAN_OBJECT_GAUGE	0x42	Unsigned integer.
SNMP_MAN_OBJECT_TIMETICKS	0x43	Time ticks
SNMP_MAN_OBJECT_OCTSTRING	0x44	Octet string.
SNMP_MAN_OBJECT_COUNTER64	0x46	64 bit counter.
SNMP_MAN_OBJECT_FLOAT	0x47	Float.
SNMP_MAN_OBJECT_DOUBLE	0x48	Double word.
SNMP_MAN_OBJECT_INTEGER64	0x4A	64 bit integer.
SNMP_MAN_OBJECT_UNSIGNED64	0x4B	64 bit unsigned integer.
SNMP_CFLAGS_KEY	0x1	Given passwords in connection data are keys and do not need key generation.
SNMP_CFLAGS_KEY_EXP_ENGIG	0x2	Given passwords should be expanded using the engine ID.

6 SNMP Manager Code Example

This example shows how to code the SNMP Manager functions and callback.

```
static void mod_snmpman_cmd ( void )
{
   t_snmp_ret snmp_ret;   /* SNMP return value */
   uint8_t    cmd_id;     /* Command ID */
   uint8_t    state;      /* Manager state */

   LOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
   state = g_smman_config.smcf_state;
   UNLOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );

   cmd_id = state & 0xF;
   snmp_ret = SNMP_SUCCESS;

   switch ( state )
   {
       case SNMPMAN_ST_CONNECT:    /* Connect */
          gio_printf( g_snmpman_gio, "SNMN connection starting %\n", snmp_ret );
          snmp_ret = snmp_connect( &g_conn_conf, &mod_snmpman_callback,
                                   &(g_smman_config.smcf_hdl) );

          LOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
          if ( g_smman_config.smcf_state == state )
          {
              if ( snmp_ret == SNMP_SUCCESS )
              {
                  g_smman_config.smcf_state = SNMPMAN_ST_WAIT_RSP;
              }
              else
              {
                  g_smman_config.smcf_state = SNMPMAN_ST_IDLE;
              }
          }
          UNLOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );

          break;
```

```
        case SNMPMAN_ST_GET:    /* Request a get command */
           snmp_ret = snmp_get_req( g_smman_config.smcf_hdl, g_oid_arr, 1U );

           LOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           if ( g_smman_config.smcf_state == state )
           {
               if ( snmp_ret == SNMP_SUCCESS )
               {
                   g_smman_config.smcf_state = SNMPMAN_ST_WAIT_RSP;
               }
               else
               {
                   g_smman_config.smcf_state = SNMPMAN_ST_CONNECTED;
               }
           }
           UNLOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           break;

        case SNMPMAN_ST_SET:    /* Request a set command */
           snmp_ret = snmp_set_req( g_smman_config.smcf_hdl, g_oid_arr, g_var_arr, 1U );

           LOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           if ( g_smman_config.smcf_state == state )
           {
               if ( snmp_ret == SNMP_SUCCESS )
               {
                   g_smman_config.smcf_state = SNMPMAN_ST_WAIT_RSP;
               }
               else
               {
                   g_smman_config.smcf_state = SNMPMAN_ST_CONNECTED;
               }
           }
           UNLOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           break;
```

Copyright HCC Embedded 2019

```
        case SNMPMAN_ST_GETNEXT:    /* Request next packet */
           snmp_ret = snmp_get_next_req( g_smman_config.smcf_hdl, g_oid_arr, 1U );

           LOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           if ( g_smman_config.smcf_state == state )
           {
               if ( snmp_ret == SNMP_SUCCESS )
               {
                   g_smman_config.smcf_state = SNMPMAN_ST_WAIT_RSP;
               }
               else
               {
                   g_smman_config.smcf_state = SNMPMAN_ST_CONNECTED;
               }
           }
           UNLOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           break;

        case SNMPMAN_ST_GETBULK:    /* Request bulk packet */
           snmp_ret = snmp_get_bulk_req( g_smman_config.smcf_hdl, g_oid_arr, 1U
                                         , g_smman_nrepeaters, g_smman_mrepeaters);

           LOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           if ( g_smman_config.smcf_state == state )
           {
               if ( snmp_ret == SNMP_SUCCESS )
               {
                   g_smman_config.smcf_state = SNMPMAN_ST_WAIT_RSP;
               }
               else
               {
                   g_smman_config.smcf_state = SNMPMAN_ST_CONNECTED;
               }
           }

           UNLOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           break;
```

```
        case SNMPMAN_ST_RECV:   /* Receive response packet */
           snmp_ret = mod_snmpman_receive();

           LOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           if ( ( g_smman_config.smcf_state == state ) && ( snmp_ret == SNMP_SUCCESS ) )
           {
               g_smman_config.smcf_state = SNMPMAN_ST_CONNECTED;
           }
           UNLOCK_NET_RESOURCE( HCC_SNMPMAN_MENU_RESID );
           break;

        case SNMPMAN_ST_DISCONNECT:   /* Disconnect */
           cmd_id = 3;
           snmp_ret = snmp_disconnect( g_smman_config.smcf_hdl );
           gio_printf( g_snmpman_gio, "SNMN Disconnected\n" );
           g_smman_config.smcf_state = SNMPMAN_ST_IDLE;
           break;
    }

    if ( snmp_ret != SNMP_SUCCESS )
    {
       if ( g_snmpman_gio != NULL )
       {
          if ( cmd_id < SNMPMAN_CMDS_CNT )
          {
              gio_printf( g_snmpman_gio
                        , "SNMN Command %s error %\n"
                        , snmpman_cmds[cmd_id].name
                        , snmp_ret );
          }
          else
          {
              gio_printf( g_snmpman_gio
                        , "SNMN Receive error %\n"
                        , snmp_ret );
          }
       }
    }
} /* mod_snmpman_cmd */
```

Copyright HCC Embedded 2019

7 Integration

This section describes all aspects of the SNMP Agent module that require integration with your target project. This includes porting and configuration of external resources.

7.1 OS Abstraction Layer

All HCC modules use the OS Abstraction Layer (OAL). This allows modules to run seamlessly with a wide variety of RTOSes, or without an RTOS.

This module uses the following OAL components:

OAL Resource	Number Required
Tasks	1
Mutexes	1
Events	1

7.2 Utilities

The SNMP Agent code creates and uses a single timer in the **hcc_timer** module.

The **hcc_timer** module is included in your system when you install the base TCP/IP modules.

Copyright HCC Embedded 2019

7.3 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer. For full details of its functions and macros, see the *HCC Base Platform Support Package User Guide*.

The SNMP modules make use of the following standard PSP functions:

Function	Package	Component	Description
psp_getrand()	psp_base	psp_rand	Generates a random number. This is described below.
psp_get_tick_count()	psp_base	psp_tick	Returns the number of milliseconds that have elapsed since the system was started. This is described below.
psp_memcmp()	psp_base	psp_string	Compares two blocks of memory.
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memmove()	psp_base	psp_string	Moves a block of memory from one location to another. The two areas of memory may overlap without this causing problems as a temporary intermediate array is used.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.

The SNMP modules make use of the following standard PSP macros:

Macro	Package	Component	Description
PSP_RD_BE16	psp_base	psp_endianness	Reads a 16 bit value stored as big-endian from a memory location.
PSP_WR_BE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as big-endian to a memory location.

psp_get_tick_count

This function is provided by the PSP to get the current tick count.

The tick resolution must be 1 ms.

Format

```
uint32_t psp_get_tick_count ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
Count	The tick count; successful execution.
SNMP_ERROR	Operation failed.

psp_getrand

This function is provided by the PSP to generate a random number.

Format

```
uint32_t psp_getrand ( uint32_t val )
```

Arguments

Argument	Description	Type
val	A value.	uint32_t

Return Values

Return value	Description
Number	The random number; successful execution.
SNMP_ERROR	Operation failed.