SNMPv3 Agent Technical Reference

Interniche Legacy Document

Version 1.00

Date: 18-May-2017 13:24

Table of Contents

1 Introduction

This Technical reference is provided with the SNMPv3 (Simple Network Management Protocol, version 3) software. The purpose of this document is to provide enough information so that a moderately experienced "C" programmer with a reasonable understanding of TCP/IP protocols can port the InterNiche SNMPv3 software to a new environment.

If the SNMPv3 code was delivered as part of an InterNiche TCP/IP stack, there is little or nothing to do. The SNMPv3 layers were compiled, linked, and tested with the IP stack. This technical reference is intended primarily as an aid to programmers porting InterNiche SNMPv3 to a non-InterNiche IP stack (or other transport).

SNMP Core (snmp directory) code is common to all SNMP agents. SNMP core implements the ASN parsing and MIB instrumentation. MIBs are compiled using a MIB compiler, and the generated source code is used by all SNMP agents. Please refer SNMPv1/SNMPv2c porting guide on how to use the MIB compiler.

1.1 Terms and Conventions

In this document, the term "stack", when used without other qualification, means the TCP/IP and related code as ported to an embedded system. "System" refers to your embedded system. "Sockets" refers to the TCP API developed for UNIX at U.C. Berkeley. A "porting engineer" refers to the engineer who is porting the SNMPv3 code. An "end user" refers to the person who ultimately ends up using the Engineer's product. "FCS" is an acronym for "First Customer Ship", the point in the software development cycle when the product is declared ready to ship. A "packet" is sequence of bytes sent on network hardware, also known as a "frame" or a "datagram".

Names of files, C structures and C routines are displayed as follows: `c_routine()`.

Samples of source code from C programs is displayed in these boxes:

```
main()
{
 printf("hello world.\n");
}
```

1.2 What SNMPv3 Is

SNMPV3 has the following enhancements over SNMPv1.

- New message format
- Security for messages
- Access control to management objects
- Remote configuration of SNMP parameters

SNMPv3 has a modular framework that supports:

- multiple SNMP versions
  - mechanisms to support SNMPv1, SNMPv2c, etc.
  - by default SNMPv3 is supported
  - InterNiche supports SNMPv1, SNMPv2c, SNMPv3
- multiple authentication and privacy algorithms
  - support for MD5 and SHA1 for authentication support for DES-CBC and CFB128-AES-128 for encryption
- multiple access models
  - View based access model used by default
  - InterNiche supports the default model.

Details about InterNiche's SNMPv3 implementation are:

1. SNMPv3 Agent based on the SNMPv3 RFCs (3411-3415, which obsolete 2571-2575)
2. View based access model for access control
3. MIB Compiler for SMIv1 and SMIv2 compliant MIBs
4. Implementation of RFC1213(MIB-II), RFC3418, RFC3411, RFC3412, RFC3413, RFC3414, and RFC3415 by default

SNMPv3 is the latest standard (RFC3411-3415), replacing the earlier standard SNMPv2. InterNiche's SNMP software supports all SNMP versions.

There are 3 versions of SNMP and these can be used in any combination.

```
#define INCLUDE_SNMPV1  1  /* SNMPv1 library, agent & hook */
#define INCLUDE_SNMPV2C 1  /* SNMPv2c (community based SNMPv2) agent */
#define INCLUDE_SNMPV3  1  /* SNMPv3 library, agent, & hook */
```

The following are other SNMP options, which are can be used with any version.

```
#define SNMP_SOCKETS    1  /* SNMP over sockets, not lightweight API */
#define PREBIND_AGENT   1  /* hardcode SNMP port into UDP */
#define INCLUDE_SNMP    1  /* update SNMP counters in TCPIP stack */
```

INCLUDE_SNMP implements SNMP core (mib instrumentation, ASN parsing, etc). It should be enabled whenever any of the above need to be used. To ensure this, following macro should be used ipport.h

```
/* if any SNMP agent is used, then INCLUDE_SNMP should be enabled */
#if (defined(INCLUDE_SNMPV1)||defined(INCLUDE_SNMPV2C)||defined(INCLUDE_SNMPV3))
#define INCLUDE_SNMP    1  /* update SNMP counters in TCPIP stack */
#endif
```

When SNMP_SOCKETS is not desired, PREBIND_AGENT should be enabled. This is done by the following macro. This is available only with InterNiche TCPIP stack.

```
#ifdef INCLUDE_SNMP
#ifndef SNMP_SOCKETS
#define PREBIND_AGENT   1  /* hardcode SNMP port into UDP */
#endif
#endif
```

1.3 What a Port Is

In the world of portable networking code, the code designer does not know what tasking system, user applications, or interfaces will be supported in the target system. So a "portable" stack is one that's designed with simple, generic interfaces in these areas, and a "glue" layer is created which maps this generic interface into the specific interfaces available on the target system. Using the example of sending a packet, the stack would be designed with a generic send_packet() call, and to porting engineer would code a "glue" routine to send the packet on the target system's network interface hardware.

Making a stack portable involves minimizing the number of calls which have to go across glue routines, and keeping the glue routines simple and therefore easy to implement. The glue routines also need to be well documented. The interfaces to the InterNiche stack have evolved through years of porting to a variety of processors, network media, and tasking systems. Wherever possible we have used standard interfaces (e. g. Sockets, ANSI C libraries) or included glue routines to illustrate their use.

The bulk of the work in porting a stack is understanding and implementing these glue routines. The InterNiche SNMPv3 has two kinds of glue routines: the first is used to interface to the IP layer, and the second to manage the SNMPv3 databases (tables, etc.).

1.4 Requirements

Before beginning a port, the programmer should ensure that the necessary resources are available in the target environment. Here is a brief summary of services InterNiche SNMPv3 needs from the system:

- A timer which ticks at least once a second.
- A non-volatile read/write method for storing database items (e.g. disk or flash memory)
- Memory as described below
- And of course, an IP stack

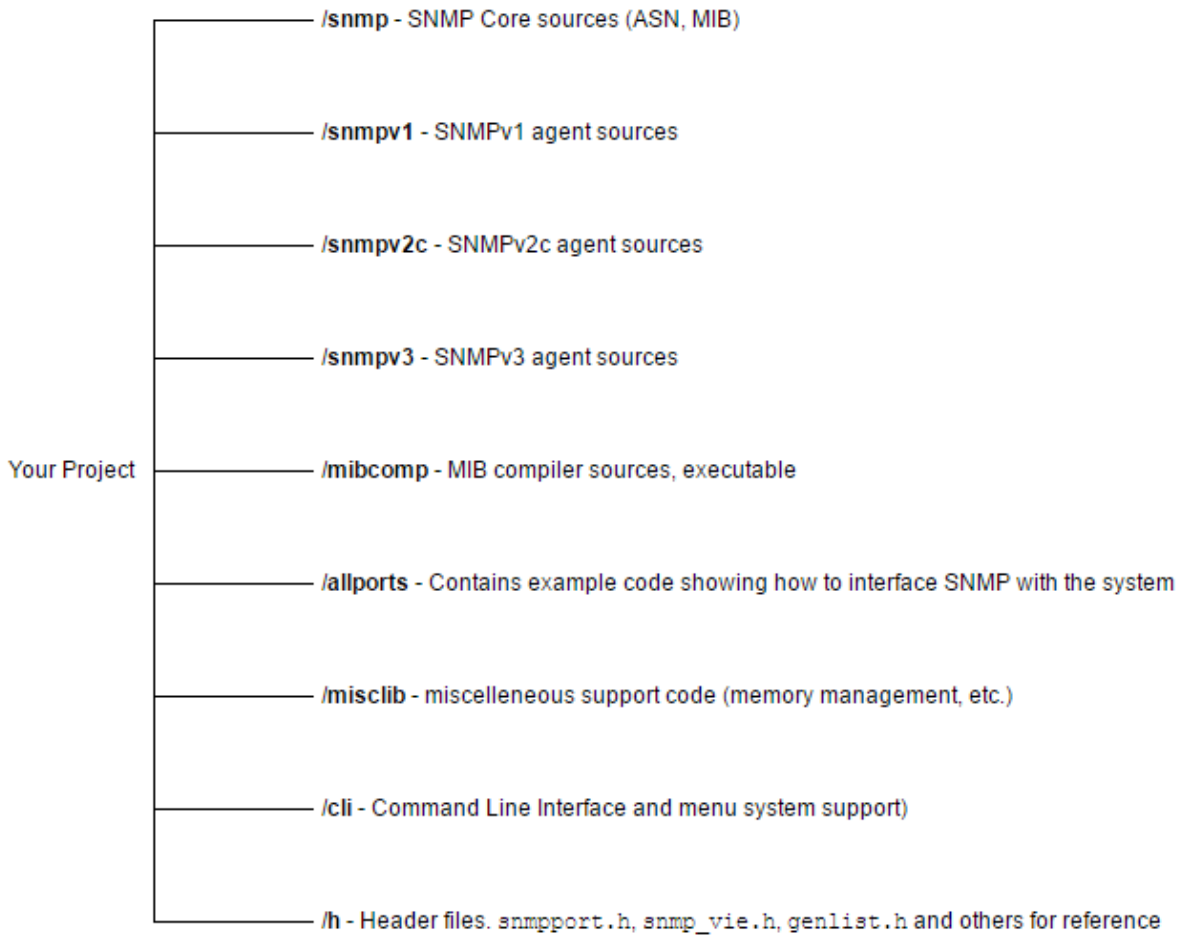
Operating System Requirements

The SNMPv3 code requires a few basic services from the Operating System. These are listed here:

clock tick	`v3_check()` needs to be called once a second to update time-since-up and boot-count
memory access	SNMPv3 obtains dynamic memory by calls to the primitives `V3_ALLOC()` and `V3_FREE()`. These can be mapped directly to the standard `calloc()` and `free()` library calls. They can also be mapped to a "partition" based system with very little effort.

1.5 SNMPv3 Source Directories List

The complete directory structure of SNMP sources is shown below.



SNMP Directory Structure

The sub-directories /snmp, /snmpv1, /snmpv2c, /snmpv3, and /mibcomp contain the InterNiche SNMP agent source distribution files. The presence/absence of snmpv1, snmpv2c and snmpv3 directories would depend on the SNMP agent(s) purchased by you. In the course of porting SNMP to your target system, you will need to copy the files from these directories to your target environment. You can rearrange the files as needed as long as you remember to set up include paths accordingly. A few of the files will need to be modified, and a few more may need to be added to support your product's particular MIBs.

2 Step By Step Porting Guide

2.1 Overview of Porting SNMPv3

This section describes the steps needed to port the InterNiche SNMPv3 to a new environment. The discussions below generally assume that the stack is being ported to a small or embedded system with a sockets API interface and that a minimal ANSI C library is available.

The recommended steps to getting the server working on your target system are as follows:

1. Copy the portable source files into your development environment.
2. Create your version of `v3port.h` and compile portable sources.
3. Code your glue layers in `v3port.c` & compile.
4. Provide hooks to SNMPv3 in your main program
5. Configure the system statically (when `V3_STATIC_TABLES` is enabled), or use the CLI for configuration.
6. Build a system, test, and debug.

SNMPv3 depends on SNMP core directory for the definition of `SnmpMib` structure and implementation of `SNMPERROR()` and `GetUpTime()`. Please refer to SNMPv1/v2 reference documentation for more details on these.

Coding Conventions

The following conventions followed in the SNMPv3 source code:

- Boolean variables have the values `TRUE` or `FALSE`. Explicit matching should be done in expressions, for example `if (v3_precalc_keys == TRUE )`.
- Functions return a value of `SUCCESS` or error number, thus to do error checking the result of a function call should be explicitly compared with `SUCCESS`, for example, `if (v3_check_sec_model (sec_model) != SUCCESS)`.

2.2 Port Dependent Files

Before beginning step one, you should be aware of which files in the InterNiche SNMPv3 distribution are the "portable" files, and which are not. The portable files are those which can be compiled and used on any target system without modification. The unportable, or "port dependent" files, are those which will need to be replaced or heavily modified for different target systems. The following is a list of SNMPv3 source files which should NOT be modified in the course of a normal port. If you feel you need to modify one of these files in the course of a routine port, please discuss it with InterNiche technical support staff first, so we can either suggest an alternative, or modify our sources to reflect the change.

2.3 Source Files List

The portable SNMPv3 source files that should not need to be modified are:

- v3app.c
- v3util.c
- v3parse.c
- misclib/genlist.c
- v3err.c
- v3usec.c
- v3cache.c
- v3main.c
- v3trap.c
- v3vacm.c
- v3mib.c
- v3vars.c
- v3app.h
- h/genlist.h
- snmp-vie.h
- v3usec.h
- v3vacm.h
- v3main.h

The network (Sockets) glue files:

- v3port.c
- snmpv3_nt.c
- v3port.h

2.4 The Master SNMPv3 Port File: v3port.h

Before you compile the source files you need to configure your version of the file `v3port.h`. This file contains most of the port dependent definitions in the stack. CPU architectures (big vs. little endian), compiler idiosyncrasies, and optional features (for example, `V3_USE_AUTH`, `V3_USE_CACHE` etc.) are controlled in this file. A single mistake in this file (such as getting big & little endian confused) will guarantee that your port won't work properly. Taking a few hours up front to implement the file line by line is time well spent. This section outlines the basic contents of `v3port.h`.

Standard Macros and Definitions

The InterNiche SNMPv3 expects `TRUE`, `FALSE` and `NULL` to be defined within the scope of v3port.h. The best way to do this is usually to include the standard C library file stdio.h inside v3port.h. If stdio.h is impractical to use or missing, the examples below will work for almost every C environment:

```
#ifndef TRUE
# define TRUE   -1
# define FALSE   0
#endif
#ifndef NULL
# define NULL (void*)0;
#endif
```

SNMPv3 also uses `SUCCESS` and `FAILURE`. If these are not defined elsewhere in your system, they can be defined as follows:

```
#define SUCCESS 0
#define FAILURE 1
```

Memory Allocation

The SNMPv3 code allocates and frees memory blocks dynamically as it runs. It uses the macros listed below to do this. If your target system supports standard C `calloc()` and `free()`, the macros map directly as follows:

```
#define V3_ALLOC(size) calloc(1,size)
#define V3_FREE(ptr)   free(ptr)
```

Many RTOS systems do not use `calloc()` due to performance issues. Generally, they use a system which supports allocations of fixed size "partitions" (blocks) instead. The macros above are designed to support this - the `V3_ALLOC()` macro only allocates a single size, (which will vary from target to target). Thus the macros can be mapped to a call to allocate the next largest partition size.

CPU Architecture

Four common macros are used from Berkeley UNIX for doing byte order conversions between different CPU architecture types. These are htons(), htonl(), ntohs(), and ntohl(). They may be either macros or functions. They accept 16 & 32 bit quantities as shown, and convert them from network format ("big-endian") to the local CPU's format.

Most IP stacks already have these byte ordering macros defined. If this is the case you should try to find the existing include file which defines them and use it rather than duplicate them. The information below is for the rare situations where these macros are not already available.

For Motorola 68000 family and most RISC chips, these can just return the variable passed.

```
#define htonl(long_var)  (long_var)
#define htons(short_var) (short_var)
#define ntohl(long_var)  (long_var)
#define ntohs(short_var) (short_var)
```

The Intel 8086 and its descendants require the byte order in the word or long to be swapped. The lswap() and bswap() primitives provided with InterNiche reference ports can be used as illustrated here:

```
#define htonl(long_var)  lswap(long_var)
#define htons(short_var) bswap(short_var)
#define ntohl(long_var)  lswap(long_var)
#define ntohs(short_var) bswap(short_var)
```

Debugging Aids

`dtrap()` is a macro called by the SNMPv3 code whenever it detects a situation which should not be occurring. The intention is for the `dtrap()` routine or macro to try to trap to whatever debugger may be in use by the programmer. Think of it as an embedded breakpoint. For most Intel x86 processor debuggers, this can be done with an `int 3` opcode. The macro below is effective if your Intel C compiler accepts inline assembly:

```
#define dtrap(); _asm{ int 3 }
```

You may need to experiment with the exact syntax to get it to compile. The stack code will generally continue executing after a `dtrap()`, but the `dtrap()`s usually indicate that something is wrong with the port. **NO PRODUCT BASED ON THIS CODE SHOULD BE SHIPPED UNTIL THE CAUSES OF ALL CALLS TO dtrap() HAVE BEEN INVESTIGATED AND RESOLVED!** When it comes time to ship code, the `dtrap()`s can be redefined to a null function to slightly reduce code size.

The next few primitives have the same function and syntax as `printf()`. They have separate names so that they can have their output redirected or be completely disabled independently of each other. The first, `dprintf()`, is used throughout the stack code to print warning messages when something seems to be wrong. This should be mapped to a debugging console or log during development, and generally `#ifdefed` away for FCS. The `gio_printf()` call is for printing statistical information from the SNMPv3 menus functions. These will certainly be useful during product development, and depending on the nature of the product may be needed in the end user's release. `gio_printf()` uses InterNiche's generic I/O mechanism. So the input/output is done with a device, for example, a console or a TELNET session. The `info_printf()` is for printing informational messages, like arrival of a packet, change of metric for a route, etc.

In most ports while the product is under development, these can both be mapped to `printf()` as shown below. Note: This example works on Microsoft C, but some compilers will complain about this syntax since it ignores the fact that these names have parameters. You may have to experiment.

```
#define dprintf     printf /* same parms as printf, but works during run time */
#define info_printf printf /* same parms as printf, used to show general info. */
```

For some products, it may make sense to define these away before FCS as follows:

```
#define dprintf      /* define to nothing */
#define info_printf  /* define to nothing */
```

The last debugging tool in `v3port.h` is the `#define NPDEBUG`. Defining this will cause the debug code to be compiled into the build. This code does things like check for valid parameters and sensible configurations during runtime. It frequently invokes `dtrap()` or `dprintf()` to inform the programmer of detected problems. You will want make sure it is defined during development. Unless ROM space is tight, it is OK to leave it defined for FCS - there will be no noticeable performance penalty from this code.

```
#define NPDEBUG 1 /* enable debug checks */
```

Features and Options

Following is a description of all the `#define` options available for SNMPv3. The most important being `V3_INICHE_IP`, which decides whether InterNiche's TCP/IP stack is used or not.

Option	Description	Dependency
`V3_AGENT` or `V3_MANAGER`	Function as SNMP Agent or SNMP Manager. **Please note** that InterNiche does not provide a complete manager implementation.	Either `V3_AGENT` or `V3_MANAGER` must be enabled.
`V3_INICHE_IP`	Use the InterNiche TCP/IP stack	Disabled if any other stack is used.

Option	Description	Dependency
INCLUDE_SNMPV3	Include SNMPV3 module. Usually it is defined in ipport.h for InterNiche's TCP/IP. For other TCP /IP stacks, it should be defined in v3port.h.	The code in all C files get enabled by INCLUDE_SNMPV3.
SNMP_SOCKETS	Use sockets API. Usually it is defined in ipport.h for InterNiche's TCP/IP. For other TCP/IP stacks, it should be defined in v3port.h.	If V3_INICHE_IP is defined, use InterNiche socket API. Otherwise use WinSock API. If SNMP_SOCKETS is disabled, PREBIND_AGENT is used.
PREBIND_AGENT	Use lightweight UDP API of InterNiche. Usually it is defined in ipport.h for InterNiche's TCP/IP. For other TCP/IP stacks, it shouldn't be used.	It is enabled if SNMP_SOCKETS is disabled. **Use only one of them**.
V3_USE_AUTH	If authentication will never be used, then code for authentication can be disabled by not using this option. Not using this option will disable code for both authentication and privacy (because privacy can never be used without authentication). This option enables: 1. code for UserTable 2. code for pre calculating keys 3. code to authenticate incoming, outgoing packets 4. code to secure (privacy) incoming, outgoing packets	If V3_USE_AUTH is not enabled then V3_USE_PRIV and V3_PRECALC_KEYS are not available even if they have been enabled separately.

Option	Description	Dependency
V3_PRECALC_KEYS	The keys for MD5 and DES are derived from the passwords (from UserTable). For MD5, auth_key is derived from auth_pwd. And for DES, priv_key is derived from priv_passwd. Since this process is very time consuming this option is provided to calculate the keys during module initialization. If V3_PRECALC_KEYS is defined, the keys are calculated at initialization time and the are_keys_valid flag is set to TRUE. At runtime, whenever a packet is to be authenticated, and the key is to be used, the are_keys_valid flag is checked for that entry. If it is FALSE, we do the calculation. Otherwise we use the previously calculated key.	If V3_USE_AUTH is disabled, then this option is not effective.
V3_USE_CACHE	Use cache to store outgoing requests. This option needs to be enabled only when request packets need to be sent. Otherwise there is no need for it.	It must be enabled if V3_MANAGER is enabled
V3_USE_NOTIFIC	Use notifications. If this is enabled, then code for sending notifications is enabled. The TADDR, TPARAM, and NOTIFY tables are enabled. So is the code to send notifications.	
V3_USE_PRIV	Use privacy. This option will enable the code that does encryption of outgoing and decryption of incoming packets.	If V3_USE_AUTH is defined, this option is effective. Otherwise this option doesn't have any effect.
V3_SHOW_ERR_MSG	Show description for error messages.	When this is disabled, only error numbers will be shown. When enabled, error numbers and description is shown.
V3_SAMPLE_CODE	This define is provided for convenience and should not be enabled. It serves as a means to include some examples with the source code. If some entries are to be added to tables at init time, then code within the V3_SAMPLE_CODE block should be copied outside the block and used.	If V3_SAMPLE_CODE is enabled, some (extra) entries will be added to the tables. As extra entries don't affect the functioning of SNMPv3, the code will still function normally.

Option	Description	Dependency
V3_STATIC_TABLES	This define should be used to statically define all the tables used for SNMPv3. Primarily to save on code size and do a build with smaller footprint. It will disable all routines used to dynamically add/del entries to the tables. Since operations associated with dynamic table modifications won't be done: • Table entries can't be modified from command line • Table entries can't be modified via SNMP SET	
V3_MIB_SUPPORT	This defines the support for SNMPv3 MIBs. Namely RFC2571-2575.	Disabled to reduce code footprint.

2.5 v3port.c

Once you have developed your `v3port.h` file as described in the previous section, the next step is to code the "glue layers".

Transport Layer

SNMPv3 has been designed to work on any transport. It has been ported and testing for TCP/IP. And it has provisions to accommodate any target. Here is how it works.

- When a packet arrives on a transport, it forms a `V3_VIRTUAL_PKT`, fills in the details about transport and calls `v3_process_rcvd_pkt()`.
- `v3_process_rcvd_pkt()` processes the packet, and if there is no error, it sends a reply using the transport information in received `V3_VIRTUAL_PKT`.
- If there is error, one of the following happens:
  - If it is a SNMPv1 packet, and if there is a SNMPv1 Message Processing Module, then the packet is passed to it.
  - If it is a SNMPv2 packet, and if there is a SNMPv2 Message Processing Module, then the packet is passed to it.
  - If there was an error processing one of the `varbinds`, then `v3_send_identical()` is called to send response reflecting the error.
  - If there was any other error, then `v3_errorpdu()` is called to form and send a `report PDU`. Since SNMPv3 discover packets don't have any varbinds, if the error is `V3_VB_NO_VBLIST`, then we have received a SNMPv3 Discovery packet, and a proper response packet is sent. If there was any other error then a `report PDU` is sent.

`V3_VIRTUAL_PKT` has a field to store type of transport. This field is used to decide on the underlying transport. The following functions use this field.

- `v3_pkt_send()`
- `v3_pkt_alloc()`
- `v3_pkt_free()`

For TCP/IP, each of the above will call the following:

- `v3_udp_send()`
- `v3_udp_alloc()`
- `v3_udp_free()`

As supplied, SNMPv3 includes code to interface with InterNiche's standard sockets or Microsoft WinSock. You need to create the routines listed below if you have another TCP/IP stack. These are described in detail in SNMPv3 Transport Layer Interface.

v3_udp_init()	Initialize UDP. It is called by `v3_init()`.
v3_udp_recv()	Callback routine for received SNMPv3 datagrams.
V3_udp_cleanup()	Clean up the data structures allocated in `v3_udp_init()`. Called by `v3_cleanup()`.
v3_udp_alloc()	Allocate a buffer for sending a SNMPv3 packet. Called by `v3_pkt_alloc()`.
v3_udp_free()	Free a buffer allocated in `v3_udp_alloc()`. Called by `v3_pkt_free()`.
v3_udp_send()	Send a SNMPv3 packet. Called by `v3_pkt_send()`.

If support for any other transport is desired, a similar set of functions should be written and called in a similar way. The receive function `v3_udp_recv()` would be called from the transport layer.

Timers and Multitasking

The following functions use the Operating System specific calls for timer ticks. They work for InterNiche's TCP/IP stack and WinSockand should be modified if you are using any other TCP/IP stack.

```
v3_start_timer()
v3_check_timeout()
v3_get_seconds_since_up()
```

The other aspect of multitasking is to protect sensitive structures from being corrupted by code re-entry. This is accomplished by two macros which protect critical sections of code. These are named `ENTER_CRIT_SECTION()` and `EXIT_CRIT_SECTION()`. On Intel systems they can usually be defines as follows:

```
#define ENTER_CRIT_SECTION()  {_asm{ pushf }; _asm{ cli } }
#define EXIT_CRIT_SECTION()   {_asm{ popf };}
```

The examples given are for a DOS port, where simple disabling interrupts for a brief period is sufficient. On a true real-time system, these should be mapped to a mutex.

2.6 SNMPv3 Static/Dynamic Data Requirements - NVRAM Parameters

The following is a list of tables and the size of each entry in the table for a recent "w32" port.

Context Table	68 bytes
Group Table	72 bytes
Access Table	172 bytes
Mibview Table	204 bytes
User Table	462 bytes
Target Address Table	166 bytes
Target Parameter Table	80 bytes
Notify Table	72 bytes
Community Table	208 bytes

For a SNMPv3 packet, a couple of memory allocations take place. One for `V3_VIRTUAL_PKT` (304 bytes) and another for storing the packet (1460 bytes).

Global Variables

SNMPv3 has a few global variables. Their initial values can be set in `v3_init()`.

- `globals.version`
- `globals.mms`
- `globals.flags`
- `globals.sec_level`
- `v3_engine_boots`
- `v3_engine_time`
- `v3_latest_rcvd_time`
- `v3_authoritative`
- `snmp_engine_id`
- `snmp_engid_len`
- `v3_precalc_keys`
- `stats`

2.7 SNMPv3 Entry Points

Once the porting of SNMPv3 has been done, you need to provide hooks for it in your main program. We will discuss regarding use of SNMPv3 over a TCP/IP transport interface. The same discussion can be easily extended to any transport. The following hooks need to be provided.

v3_init()	needs to be called during initialization time, after the TCP/IP stack has been initialized.
v3_cleanup()	needs to be called when the program is quitting, so that all the dynamic memory used by SNMPv3 can be freed.
v3_check()	needs to be called every clock-tick, so that it can update its timers.
v3_udp_recv()	needs to be called whenever a packet is received on the SNMP port numbers (port number 161, 162). For InterNiche SNMPv3, this hook is provided in the `udpdemux()` routine of file `udp.c`. If another TCP/IP stack is used, then it is advisable to open a socket to listen to these port numbers in `v3_udp_init()` of `v3port.c`.

2.8 Testing

Once your `v3port.h` file is set up and your glue layers are coded, compiled, and linked, you are ready to test your SNMPv3. Although testing SNMPv3 is a rigorous process InterNiche's protocol implementation is quite amenable to source level debugging with breakpoints. Setting a breakpoint on `v3_udp_recv()` will allow you to trace the entire processing of a received SNMPv3 packet.

To test InterNiche's SNMPv3 agent implementation, we recommend using any of the freely available packet analysis and SNMPv3 Managers easily found with a web search.

3 Additional Authentication and Privacy Algorithms

InterNiche SNMPv3 implementation is very generic and facilitates adding support for other protocols for authentication or privacy. This section discusses how it can be done.

3.1 Supporting a New Authentication Algorithm

To add support for a new Authentication protocol you need to do the following things. For illustration, we show the source code for the MD5 protocol which is shipped with SNMPv3.

Implementing the Three Essential Functions

One function converts the password to the key, and the other two authenticate the incoming message and the outgoing message. For example, the following declarations in `v3port.h` demonstrate these MD5 function implementations.

```
#ifdef V3_USE_AUTH
void v3_password_to_key_md5(u_char *password, unsigned passwordlen,
        u_char *engineID, unsigned engineLength, u_char *key);
int  v3_auth_outgoing_md5(u_char *auth_key, unsigned key_len,
   u_char *auth_start,u_char *pkt_start,unsigned *len );
int  v3_auth_incoming_md5(u_char *auth_key, unsigned key_len,
   u_char *auth_start,u_char *pkt_start,unsigned *len );
#endif /* V3_USE_AUTH */
```

Defining a Name for the Algorithm

The name is defined in v3port.h. For example:

```
#define MD5NAME "MD5"
```

Adding the Algorithm Object-Identifier (OID)

The algorithm object identifier can be added via the CLI or `v3_auth_tbl_init()`. The following command in the SNMPv3 CLI script file adds an OID for the new algorithm. The name should be same as that defined in the previous step.

```
snmpv3 authoid -n MD5 -o 1.3.6.1.6.3.10.1.1.2
```

If the protocol is to be added programmatically, then the programmer should do something similar to the `auth_entry` variable in `v3_auth_tbl_init()` as follows.

```
int
v3_auth_tbl_init()
{
   /* Struct used to add an entry for MD5 authentication algorithm at
      init time. Included here as an example, because this entry also
      gets added via the "snmpv3 authoid" CLI command.

   /* add a static entry to the user table */
   /* niche_add(p_auth_list,(GEN_STRUCT)&auth_entry);  */

   return SUCCESS;
}
```

Mapping Algorithm Names to Corresponding Functions

This mapping is done in the function `v3_auth_prot_init()` of `v3port.c`. The following code example does the mapping for MD5.

```
auth_entry=(AUTH_TABLE)niche_lookup_name(p_auth_list,MD5NAME);

if ( auth_entry == NULL ) /* entry not found for MD5 */
   return FAILURE ;

auth_entry->pwd_to_key = v3_password_to_key_md5 ;
auth_entry->incoming.auth = v3_auth_incoming_md5 ;
auth_entry->outgoing.auth = v3_auth_outgoing_md5 ;
```

3.2 Supporting a New Privacy Algorithm

Supporting a new Privacy protocol is similar to the steps for supporting a new Authentication protocol. The following things need to be done to add support for a new Privacy protocol. For illustration, we will show the source code for the DES protocol, shipped with SNMPv3.

Implementing the Two Essential Functions

One function is needed to decrypt an incoming message and another to encrypt an outgoing message. For example, the following declarations in `v3port.h` demonstrate these DES function implementations.

```
#ifdef V3_USE_PRIV
int  v3_priv_outgoing_des(
        u_char *key   ,unsigned key_len,
        u_char *iv    ,unsigned iv_len,
        u_char *plain ,unsigned plain_len,
        u_char *cipher,unsigned *cipher_len);
int  v3_priv_incoming_des(
        u_char *key   ,unsigned key_len,
        u_char *iv    ,unsigned iv_len,
        u_char *cipher,unsigned cipher_len,
        u_char *plain ,unsigned plain_len);
#endif /* V3_USE_PRIV */
```

Defining a Name for the Algorithm

The name is defined in `v3port.h`. For example:

```
#define DESNAME "DES"
```

Adding the Algorithm Object-Identifier (OID)

The algorithm object identifier can be added via the CLI or `v3_auth_tbl_init()`. The following command in the SNMPv3 CLI script file adds an OID for the new algorithm. The name should be same as that defined in the previous step.

```
snmpv3 authoid -n DES -o 1.3.6.1.6.3.10.1.2.2
```

If the protocol is to be added programmatically, then the programmer should do something similar to the auth_entry variable in `v3_auth_tbl_init()` as follows.

The following code adds an entry for DES.

```
int
v3_auth_tbl_init()
{
   /* Struct used to add an entry for DES encryption algorithm at
      init time. Included here as an example, because this entry also
      gets added via the "snmpv3 authoid" CLI command.

      struct AuthTable auth_entry = { 0, DESNAME, {{1,3,6,1,6,3,10,1,2,2},10}};
   */

   niche_list_constructor(p_auth_list,sizeof(struct AuthTable));

   /* add a static entry to the user table */
   /* niche_add(p_auth_list,(GEN_STRUCT)&auth_entry);  */

   return SUCCESS;
}
```

Mapping Algorithm Names to Corresponding Functions

This mapping is done in the function `v3_auth_prot_init()` of `v3port.c`. The following code example does the mapping for DES.

```
auth_entry=(AUTH_TABLE)niche_lookup_name(p_auth_list,DESNAME);

if ( auth_entry == NULL ) /* entry not found for MD5 */
   return FAILURE ;

auth_entry->pwd_to_key = NULL ;
auth_entry->incoming.priv = v3_priv_incoming_des ;
auth_entry->outgoing.priv = v3_priv_outgoing_des ;
```

For a user, the `pwd_to_key` algorithm for DES is same as that used for authentication. Hence, the `pwd_to_key` field is made `NULL`.

4 The SNMPv3 User Menu

The SNMPv3 module comes with portable C code to implement a few simple diagnostic commands on command line interface. The commands can be invaluable both during debugging SNMPv3 and to the end user during configuration and runtime. If you do not implement these menu commands as provided (can be done by disabling V3_USE_MENU), we strongly suggest that some alternative method (i.e. a GUI) be provided to the end user for accessing the same data.

The menu commands are described below:

4.1 snmpv3 access

Command Name

`snmpv3 access - manage the SNMPv3 access table`

Syntax

```
access -m INT -g STRING -l INT -c STRING [-x] -r STRING -w STRING -n STRING
[-t INT -s INT]
```

```
access -d INT
```

Parameters

`-m`	Argument of type `INT`, specifying `the security model of the SNMPv3 access table entry`.
`-g`	Argument of type `STRING`, specifying `the group name of the SNMPv3 access table entry`.
`-l`	Argument of type `INT`, specifying `the security level of the SNMPv3 access table entry`.
`-c`	Argument of type `STRING`, specifying `the context name associated with the SNMPv3 access table entry`.
`-x`	(no parameter), specifies that an exact match of `the context name is required`.
`-r`	Argument of type `STRING`, specifying `the read view name`.
`-w`	Argument of type `STRING`, specifying `the write view name`.
`-n`	Argument of type `STRING`, specifying `the notify view name`.
`-t`	Argument of type `INT`, specifying `the storage type of the SNMPv3 table entry`.
`-s`	Argument of type `INT`, specifying `the row status of the SNMPv3 table entry`.
`-d`	Argument of type `INT`, specifying `the index of the SNMPv3 access table entry to delete`.

Description

This command manages the SNMPv3 access table.

If the command is successful, the updated table is displayed.

Notes/Status

- If the context string is too long, it is truncated.
- Table indexes begin at 1.
- Possible values for the access modes are: `RONLY, RWRITE, NOACCESS, READCREATE, WRITEONLY, ACCESSIBLE_FOR_NOTIFY, and NOTIFY`.
- NOTIFY is equivalent to ACCESSIBLE_FOR_NOTIFY.
- The `'-t'` parameter defaults to 4 = permanent.
- The `'-s'` parameter defaults to 1 = active.
- The `'-d'` parameter takes precedence over all other parameters.

Location

This command is provided by the `SNMP` module when `INCLUDE_SNMP` and `INCLUDE_SNMPV3` are defined.

4.2 snmpv3 authoid

Command Name

`snmpv3 authoid - manage the SNMPv3 algorithm table`

Syntax

```
authoid -n STRING -o STRING

authoid -d INT
```

Parameters

`-n`	Argument of type `STRING`, specifying `the name of authentication or encryption algorithm`.
`-o`	Argument of type `STRING`, specifying `the OID of the authentication or encryption algorithm`.
`-d`	Argument of type `INT`, specifying `the index of the entry to delete`.

Description

This command manages the auth-oid table.

If the command is successful, the updated table is displayed.

Notes/Status

- Table entry indexes start at 1.
- If the name string is too long, it is truncated.
- The `'-d'` parameter takes precedence over all other parameters.

Location

This command is provided by the `SNMP` module when `ENABLE_SNMP_TRAPS` is defined.

4.3 snmpv3 context

Command Name

`snmpv3 context - manage the SNMPv3 context table`

Syntax

```
context -n STRING

context -d INT
```

Parameters

`-n`	Argument of type `STRING`, specifying the context name to add to the SNMPv3 context table.
`-d`	Argument of type `INT`, specifying the index of the SNMPv3 context table entry to delete.

Description

This command manages the SNMPv3 context table entries.

If the command is successful, the updated table is displayed.

Notes/Status

- If the context string is too long, it is truncated.
- The context indices are zero-based.

Location

This command is provided by the `SNMP` module when `INCLUDE_SNMP` is defined.

4.4 snmpv3 group

Command Name

`snmpv3 group - manage the SNMPv3 group table`

Syntax

```
group -g STRING -n STRING -m INT [-t INT] [-s INT]

group -d INT
```

Parameters

`-g`	Argument of type `STRING`, specifying `the group name`.
`-n`	Argument of type `STRING`, specifying `the security name`.
`-m`	Argument of type `INT`, specifying `the security model of the SNMPv3 group table entry`.
`-t`	Argument of type `INT`, specifying `the storage type of the SNMPv3 table entry`.
`-s`	Argument of type `INT`, specifying `the row status of the SNMPv3 table entry`.
`-d`	Argument of type `INT`, specifying `the index of the SNMPv3 group table entry to delete`.

Description

This command manages the group table, which maps group names to security models.

If the command is successful, the updated table is displayed.

Notes/Status

- If the name string is too long, it is truncated.
- Table entry indexes start at 1.
- The `'-t'` parameter defaults to 4 = permanent.
- The `'-s'` parameter defaults to 1 = active.
- The `'-d'` parameter takes precedence over all other parameters.

Location

This command is provided by the `SNMP` module when `ENABLE_SNMP` and `INCLUDE_SNMPV3` are defined.

4.5 snmpv3 mibview

Command Name

`snmpv3 mibview - manage the SNMPv3 mibview table`

Syntax

```
mibview -n STRING -o STRING -m STRING [-x] [-t INT] [-s INT]

mibview -d INT
```

Parameters

`-n`	Argument of type `STRING`, specifying `the name of the MIB view`.
`-o`	Argument of type `STRING`, specifying `the OID associated with the MIB view`.
`-m`	Argument of type `INT`, specifying `the OID mask`.
`-x`	(no parameter), exclude this MIB subtree from the MIB view.
`-t`	Argument of type `INT`, specifying `the storage type of the SNMPv3 table entry`.
`-s`	Argument of type `INT`, specifying `the row status of the SNMPv3 table entry`.
`-d`	Argument of type `INT`, specifying `the index of the SNMPv3 table entry to delete`.

Description

This command manages the MIB view table.

If the command is successful, the updated table is displayed.

Notes/Status

- If the name string is too long, it is truncated.
- Table entry indexes start at 1.
- The `'-t'` parameter defaults to 4 = permanent.
- The `'-s'` parameter defaults to 1 = active.
- The `'-d'` parameter takes precedence over all other parameters.

Location

This command is provided by the `SNMP` module when `ENABLE_SNMP` and `INCLUDE_SNMPV3` are defined.

4.6 snmpv3 notify

Command Name

`snmpv3 notify - manage the SNMPv3 notify table`

Syntax

```
notify -m INT -n STRING -v STRING [-t INT] [-s INT]

notify -d INT
```

Parameters

`-m`	Argument of type `INT`, specifying `the notify type`.
`-n`	Argument of type `STRING`, specifying `the notify group name`.
`-o`	Argument of type `STRING`, specifying `the notify tag`.
`-t`	Argument of type `INT`, specifying `the storage type of the SNMPv3 table entry`.
`-s`	Argument of type `INT`, specifying `the row status of the SNMPv3 table entry`.
`-d`	Argument of type `INT`, specifying `the index of the SNMPv3 table entry to delete`.

Description

This command manages the Notify table used in sending SNMPv3 notification messages.

If the command is successful, the updated table is displayed.

Notes/Status

- If the name string is too long, it is truncated.
- Table entry indexes start at 1.
- The `'-t'` parameter defaults to 4 = permanent.
- The `'-s'` parameter defaults to 1 = active.
- The `'-d'` parameter takes precedence over all other parameters.

Location

This command is provided by the `SNMP` module when `ENABLE_SNMP` and `INCLUDE_SNMPV3` are defined.

4.7 snmpv3 tables

Command Name

`snmpv3 tables - display all SNMPv3 tables`

Syntax

`tables`

Description

Displays all of the SNMPv3 tables.

Location

This command is provided by the `SNMP` module when `INCLUDE_SNMP` and `INCLUDE_SNMPV3` are defined.

4.8 snmpv3 taddr

Command Name

`snmpv3 taddr - manage the SNMPv3 target address table`

Syntax

```
taddr -n STRING -a IPADDR -v STRING -r INT -x INT -z STRING [-t INT] [-s
INT]
```

```
taddr -d INT
```

Parameters

`-n`	Argument of type `STRING`, specifying `the entry name`.
`-a`	Argument of type `IPADDR`, specifying `the destination IP address`.
`-v`	Argument of type `STRING`, specifying `the trap tag value`.
`-r`	Argument of type `INT`, specifying `the retry count`.
`-x`	Argument of type `INT`, specifying `the timeout (seconds)`.
`-z`	Argument of type `STRING`, specifying `the trap parameter(s)`.
`-t`	Argument of type `INT`, specifying `the storage type of the SNMPv3 table entry`.
`-s`	Argument of type `INT`, specifying `the row status of the SNMPv3 table entry`.
`-d`	Argument of type `INT`, specifying `the index of the SNMPv3 table entry to delete`.

Description

This command manages the Target Address table.

If the command is successful, the updated table is displayed.

Notes/Status

- If the name string is too long, it is truncated.
- Table entry indexes start at 1.
- The `'-t'` parameter defaults to 4 = permanent.
- The `'-s'` parameter defaults to 1 = active.
- The `'-d'` parameter takes precedence over all other parameters.

Location

This command is provided by the SNMP module when ENABLE_SNMP and INCLUDE_SNMPV3 are defined.

4.9 snmpv3 tparam

Command Name

`snmpv3 tparam - manage the SNMPv3 target parameters table`

Syntax

```
tparam -m INT -n STRING -u INT -v STRING -l INT [-t INT] [-s INT]

tparam -d INT
```

Parameters

`-m`	Argument of type `INT`, specifying `the MPC model of the trap`.
`-n`	Argument of type `STRING`, specifying `the name of this entry`.
`-u`	Argument of type `INT`, specifying `the USEC model`.
`-v`	Argument of type `STRING`, specifying `the Security model name`.
`-l`	Argument of type `INT`, specifying `the Security level`.
`-t`	Argument of type `INT`, specifying `the storage type of the SNMPv3 table entry`.
`-s`	Argument of type `INT`, specifying `the row status of the SNMPv3 table entry`.
`-d`	Argument of type `INT`, specifying `the index of the SNMPv3 table entry to delete`.

Description

This command manages the Target Prameter table. If the command is successful, the updated table is displayed.

Notes/Status

- If the name string is too long, it is truncated.
- Table entry indexes start at 1.
- The `'-t'` parameter defaults to 4 = permanent.
- The `'-s'` parameter defaults to 1 = active.
- The `'-d'` parameter takes precedence over all other parameters.

Location

This command is provided by the module when `ENABLE_SNMP` and `INCLUDE_SNMPV3` are defined.

4.10 snmpv3 username

Command Name

`snmpv3 username - manage the SNMPv3 user table`

Syntax

```
username -u STRING -v STRING -a STRING -b STRING -p STRING -q STRING [-t
INT] [-s INT]
```

```
username -d INT
```

Parameters

`-u`	Argument of type `STRING`, specifying `the user name`.
`-v`	Argument of type `STRING`, specifying `the Security name`.
`-a`	Argument of type `STRING`, specifying `the Authentication entry in the Auth-OID table`.
`-b`	Argument of type `STRING`, specifying `the Authentication password`.
`-p`	Argument of type `STRING`, specifying `the Privacy entry in the Auth-OID table`.
`-q`	Argument of type `STRING`, specifying `the Privacy password`.
`-t`	Argument of type `INT`, specifying `the storage type of the SNMPv3 table entry`.
`-s`	Argument of type `INT`, specifying `the row status of the SNMPv3 table entry`.
`-d`	Argument of type `INT`, specifying `the index of the SNMPv3 table entry to delete` .

Description

This command manages the User table.

If the command is successful, the updated table is displayed.

Notes/Status

- If the name string is too long, it is truncated.
- Table entry indexes start at 1.
- The `'-t'` parameter defaults to 4 = permanent.
- The `'-s'` parameter defaults to 1 = active.
- The `'-d'` parameter takes precedence over all other parameters.

Location

This command is provided by the SNMP module when ENABLE_SNMP and INCLUDE_SNMPV3 are defined.

4.11 snmpv3 v3test

Command Name

`snmpv3 v3test - test SNMPv3 security algorithm`

Syntax

```
v3test (-a | -m)
```

Parameters

`-a`	(no parameter), perform all authentication tests.
`-m`	(no parameter), perform MD5 authentication test.

Description

This command tests the security algorithms supported by the build.

Notes/Status

- This commands only supports the MD5 authentication algorithm.

Location

This command is provided by the `SNMP` module when `INCLUDE_SNMP` and `V3_USE_AUTH` are defined.

5 MIB Compiler for SMIv1 and SMIv2 MIBs

5.1 MIB Compiler Introduction

SNMPv3 uses MIBs which are written using the definitions of SMIv2 (Structure of Management Information, version 2). SNMPv1 used MIBs written in SMIv1 format. The MIB compiler of InterNiche supports all MIBs.

5.2 Feature Additions to MIB Compiler for SMIv2

This section mainly describes features that were added to the MIB compiler so that it can parse MIBs of SMIv2 format also. RFC2578, RFC2579, and RFC2580 were used as a reference.

RFC2578 describes new macros for module definitions, object definitions, and notification definitions. The following macros from RFC2578 are implemented:

- `MODULE-IDENTITY`
- `OBJECT-IDENTITY`
- `OBJECT-TYPE`
- `NOTIFICATION-TYPE`

The file `rfc2578.mib` is shipped with the MIB compiler and it contains the MIB definitions of RFC2578.

RFC2579 describes the various textual conventions that would be used with SMIv2 MIBs. The file `rfc2579.mib` is shipped with the MIB compiler, and it contains the MIB definitions of RFC2579. The `TEXTUAL-CONVENTION` macro has been implemented in the MIB compiler to parse all the textual-conventions in SMIv2 MIBs.

RFC2580 contains "Conformance Statements for SMIv2". The following macros from this RFC have been implemented:

- `OBJECT-GROUP`
- `NOTIFICATION-GROUP`
- `MODULE-COMPLIANCE`
- `AGENT-CAPABILITIES`

For compiling SMIv2 MIBs, it is advised that `rfc2578.mib` and `rfc2579.mib` should also be compiled.

5.3 Compile Time Options for the MIB Compiler

The MIB compiler has the following options which can be enabled or disabled in the file `parse.h`.

SMIV2	Enable the parsing of SMIv2 MIBs
NPDEBUG	Enable debugging for memory allocations
V2_TRAP_SRC	Enable the generation of source code for traps (`NOTIFICATION-TYPE` macros). Effective only if SMIV2 is enabled.
V2_TOKEN_SRC	Enable the generation of source code for user-defined data types (`TEXTUAL-CONVENTION` macros). Effective only if SMIV2 is enabled.

5.4 Row Creation and Deletion

The MIB compiler doesn't do anything special for facilitating row creation and deletion in tables. It generates stubs the same way it does for SMIv1. One of the main reasons for this is that each table is normally implemented separately, and has its own means to access values, do lookups, insert rows, delete rows etc. In such a scenario, it is not possible to generate generic code.

The stub function generated for each table should take care of row creation and deletion. As an example, SNMPv3 is shipped with row creation and deletion facility for `vacmSecurityToGroupTable` of RFC2575. Please refer to the SNMPv1/v2c manual for a detailed description on handling `GET`, `GETNEXT`, and `SET` for the objects of a table. The description in this section is a step further in that direction.

Normal GET, GETNEXT, SET for vacmSecurityToGroupTable

When a `GET`, `GETNEXT`, or `SET` request comes for an `OID` belonging to `vacmSecurityToGroupTable`, the function `var_vacmSecurityToGroupEntry()` is called.

For a `GET` operation, the following need to done:

1. Compare the requested `OID`, with `OID`s of the existing table, and finding a match. If a match is not found, return `NULL`.
2. Return a pointer to the value for this `OID`

To accomplish this, we do the following:

1. For each row of the existing table, form an `OID` with an index for that row.
2. Compare it with the received `OID`.
3. Do this until a match occurs or we reach the end of the table. If we reached the end of the table, then return `NULL`.
4. For the matching row entry, return the pointer to location where value is stored.
5. Update the name and length arguments to contain the `Object Identifier` for the accessed value, and the length of `Object Identifier`. For a `GET` operation this is the same as that received. Also `var_len` argument should be updated to contain the length of the value accessed by the incoming `OID`.

For a `SET`, we do the following as well:

1. If we are using a special function to do `SET`, the global variable `set_parms.access_method` needs to be updated to point to it.
2. For a string, `var_len` should contain the maximum length of the string.
3. To force length checking for a string the following globals need to be updated:
   - `set_parms.do_range`
   - `set_parms.hi_range`
   - `set_parms.lo_range`
4. If an access_method is used, then it should be implemented.

With GETNEXT we need to do an enhanced search for OID. For all rows in the table, we select the next one lexicographically.

The following source code from v3mib.c does the things discussed above. The code for row creation and deletion is omitted from this example and will be added after we have discussed that. This table is indexed by security-model and security-name.

```
u_char *
var_vacmSecurityToGroupEntry(
  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 */
{
   oid        lowest[DEF_VARLEN*3];      /* "best fit" object Id */
   oid        current[DEF_VARLEN*3];
   u_char *   cp;
   int        i,len,oid_index,index,ind_select=-1,len_select;
   struct GroupTable *entry;

   /* Get the number of rows in the table */
   len = niche_list_len(p_group_list);
   *var_len = sizeof(long);   /* default length */

   /* fill in object part of name for current (less sizeof instance part) */
   MEMCPY((char *)current, (char *)vp->name, vp->namelen * sizeof(oid) );

   oid_index = vp->namelen ;
   /* scan at table for closest match */
   for (index=0 ; index<len ; index++ )
   {
      /* Add the 1st index to the OID */
      entry = (GRP_TABLE)niche_list_getat(p_group_list,index);
      current[oid_index-1] = entry->sec_model;

      /* Add the 2nd index to the OID */
      cp = (u_char *)entry->sec_name ;
      /* create object id reflecting this table entry's name*/
      current[oid_index] = strlen((char *)cp);

      for(i = 0; i < (int)current[oid_index] ; i++)
         current[oid_index+i+1] = *cp++;

      if (oper)   /* operation is SET or GET */
      {
         if (compare(current, oid_index+i+1, name, *length) == 0)
         {
            /* Found an exact match */
            len_select=oid_index+i+1;
            MEMCPY((char *)lowest, (char *)current,
               len_select * sizeof(oid));
            ind_select=index;
            break;   /* no need to search further */
         }
      }
```

```
      else   /* caller wants closest match */
      {
         /* if new one is greater than input and closer to input than
         previous lowest, save this one as the "next" one. */
         if ( (compare(current, oid_index+i+1, name, *length) > 0) &&
            ( ((ind_select==-1) ||
                compare(current, oid_index+i+1, lowest, oid_index+i+1) < 0) ) )
         {
            len_select=oid_index+i+1;
            MEMCPY((char *)lowest, (char *)current, len_select * sizeof(oid));
            ind_select=index;
         }
      }

   } /* end for loop */

   if ( ind_select == -1 ) /* could not find entry in context table */
   {
      return NULL ;  /* entry not found for GET/GETNEXT request */
   }

   *length = len_select;
   MEMCPY((char *)name, (char *)lowest, (*length) * sizeof(oid));

   entry = (GRP_TABLE)niche_list_getat(p_group_list,ind_select);

   switch(vp->magic)
   {
   case VACMSECURITYMODEL:
      long_return = entry->sec_model ;
      return (u_char *)&long_return;
   case VACMSECURITYNAME:
      *var_len = strlen(entry->sec_name);
      return (u_char *)entry->sec_name;
   case VACMGROUPNAME:
      if(oper == SET_OP)
         *var_len = SNMP_NAME_LEN;
      else
         *var_len = strlen(entry->group_name);
      return (u_char *)entry->group_name;
   case VACMSECURITYTOGROUPSTORAGETYPE:
      long_return = entry->storage_type ;
      return (u_char *)&long_return;
   case VACMSECURITYTOGROUPSTATUS:
      long_return = entry->status ;
      return (u_char *)&long_return;
   default:
     SNMPERROR("var_AtEntry: bad magic number");
   }

   return NULL; /* default FAIL return.*/
}
```

Row Deletion for vacmSecurityToGroupTable

For row deletion, the table has a member with datatype `RowStatus`. For `vacmSecurityToGroupTable`, the member is `vacmSecurityToGroupStatus`. When a SET value of 6 (destroy) is received for this object, the corresponding row has to be deleted. To do this, you will have to do the following.

1. Have a special method to do SET for `vacmSecurityToGroupStatus`. For all other members, the default SET mechanism is used.
2. For a SET request to `vacmSecurityToGroupStatus`, `var_vacmSecurityToGroupEntry()` is called first. Traditionally this sets the `access_method` and returns a pointer to the variable whose value will be changed by the `access_method` For this case, we will SET the `access_method` to `set_vacmSecurityToGroupEntry`, and return a pointer to the entry (the corresponding row), instead of the status member of this row.
3. When `set_vacmSecurityToGroupEntry()` is called, it will check the received value. If this value is 6, we delete the row.

The following lines of `var_vacmSecurityToGroupEntry()` are updated.

```
case VACMSECURITYTOGROUPSTATUS:
   if(oper == SET_OP)
   {
      set_parms.access_method = (int (*)())set_vacmSecurityToGroupEntry;
      return (u_char *)entry ;
   }
   else
   {
      long_return = entry->status ;
      return (u_char *)&long_return;
   }
```

Following is the new function `set_vacmSecurityToGroupEntry()`.

```
int
set_vacmSecurityToGroupEntry(u_char  *var_val, /* pointer to asn1 encoded set value ??? */
   u_char  var_val_type,   /* asn1 type of set value */
   int      var_val_len,   /* length of set value */
   u_char  *statP,         /* pointer returned by var_atEntry */
   int      statLen)       /* *var_len" from var_atEntry */
{
    /* statP points to an entry in table and not to the variable
       corresponding to an OID. This function gets called for
       sets to vacmSecurityToGroupsStatus.

       Check the value. If it is 6 (destroy), delete the entry.
       Else, update the status field */

   struct GroupTable *entry= (GRP_TABLE)statP;
   int status ;

   setVariable(var_val, var_val_type, (u_char *)&status, statLen);
   if ( status == 6 )   /* 6 is destroy */
      niche_del(p_group_list,entry);
   else
   {
      if ( entry->status = 5 ) /* create-and-wait */
         entry->status = 2 ;
      else if ( entry->status = 4 ) /* create-and-go */
      {
   /* if grp_name is not set, then we are not ready */
   /* Try createAndGo again, after setting grp_name */
         if ( strlen(entry->group_name) == 0 )
            entry->status = 3; /* notReady */
         else
            entry->status = 1; /* active */
      }
      else
         entry->status = status ;
   }
```

Row Creation for vacmSecurityToGroupTable

When we get a SET request for a row that isn't there, we treat that as a request for a new row. The `var_varmSecurityToGroupEntry()` function creates a new row and sets the default values.

Depending on the create-and-go request or create-and-wait request for row creation, the status of the new row is set. This checking is done in set_vacmSecurityToGroupEntry() listed in the previous section. The code which reflects this is in `var_vacmSecurityToGroupEntry()` as follows.

```
if ( ind_select == -1 ) /* could not find entry in context table */
{
   /* For a SET operation, treat this as a request for row creation */
   if(oper == SET_OP)
   {
      /* Create a new row here */
      struct GroupTable grp_entry = { 0, "", "", 4,3 } ;
      grp_entry.sec_model= name[oid_index-1];
      for ( i=0 ; i < (int)name[oid_index]; i++ )
         grp_entry.sec_name[i]=(u_char)name[oid_index+i+1];
      grp_entry.sec_name[i]=0;

      niche_add(p_group_list,(GEN_STRUCT)&grp_entry);
      ind_select = 0 ;
   }
   else
      return NULL ;  /* entry not found for GET/GETNEXT request */
}
```

Similar implementations can be done for each of the tables for which row-creation or row-deletion is desired.

5.5 Organization of Files Generated by the MIB Compiler

The MIB Compiler parses a set of MIB files and generates sources files depending on the command line options used. When all four options, `-chvn`, are used, four files are generated.

1. A `.c` file containing stubs for groups in all the parsed MIBs. The name of this file is picked up from the `DEFINITIONS` line of the last MIB parsed. For example, 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.
2. A `.h` file containing declarations for the stub. The name would be `snmp-vie.h`.
3. `snmpvars.c` contains a list of all the `OID`s in the MIBs.
4. `snmp-vie.num` is just for additional information.

We suggest that the files `snmpvars.c`, `snmp-vie.h` be placed in the SNMP folder. And that the implementations of functions (prototypes in `snmp-vie.c`) be placed in implementation directories. For example, the following scheme is used for InterNiche's SNMP implementation.

1. The RFCs that are compiled are `rfc213.mib` (MIB-2), `rfc2578.mib` (SNMPv3 basic subtrees), `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 SNMP folder.
4. The implementation for RFC1213 is put in file `rfc1213.c` in SNMP folder.
5. The implementations for the rest of the MIBs are SNMPv3 dependent and put in the file `v3mib.c` in SNMPV3 folder.

5.6 Implementing TestAndIncr Objects

Definition

The TestAndIncr object is defined in RFC2579 as "represents integer-valued information used for atomic operations". Objects defined as this type allow only one management station to modify the contents at any point in time. Here is how they behave

- On a GET, GETNEXT, and GETBULK, the current value is incremented and the new value is returned.
- On a SET, the received value is checked with the current value. If they are the same, then SET is allowed for rest of the variables in the PDU. Otherwise, this SET, and the SET for remaining variables is not allowed. In a SET operation, the value is not incremented (this behavior can be changed).

InterNiche implementation has a limitation that the TestAndIncr object controls only the remaining varbinds. That is, in the PDU of an SNMP SET message, if the TestAndIncr object is number 4 in a list of 10, and if the comparison for TestAndIncr object fails, then SET for objects 5 to 10 won't be allowed. But the SET for objects 1 through 3 would already have taken place. So, when TestAndIncr object is being SET, it is advised that it should be the first varbind in the PDU.

Example

To illustrate how `TestAndIncr` objects can be implemented, we will take the example of `usmUserSpinLock` object of RFC2574. The implementation for `GET`, `GETNEXT`, `GETBULK` operations will increment the variable and then return the value. For a `SET` operation, the function `set_usmUser()` will be called. Here is code added to the `var_usmUser()` generated by the MIB Compiler.

```
u_char *
var_usmUser(
   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 */
{
   if(oper && (compare(name, *length, vp->name, (int)vp->namelen) != 0))
      return NULL;

   MEMCPY(name, vp->name, (int)vp->namelen * sizeof(oid));
   *length = vp->namelen;
   *var_len = sizeof(long);   /* default length */

   switch (vp->magic)
   {
   case USMUSERSPINLOCK :
      if ( oper == SET_OP )
      {
         set_parms.access_method = set_usmUser;
         return (u_char *)&stats.usmUser.usmUserSpinLock;
      }
      else
      {
         long_return = ++stats.usmUser.usmUserSpinLock; /* Inc. and send */
         return (u_char *)&long_return;
      }
   default:
       SNMPERROR("var_usmUser: Unknown magic number");
   }
   return NULL;   /* default FAIL return. */
}
```

For a SET operation, the `access_method` is set to `set_usmUser` and a pointer to the variable for `usmUserSpinLock` is returned. For other operations, the value is incremented and returned.

In `set_usmUser()`, the received value is compared with the current value. Here is the sample code.

```
int
set_usmUser(u_char  *var_val, /* pointer to asn1 encoded set value ??? */
   u_char  var_val_type,   /* asn1 type of set value */
   int      var_val_len,   /* length of set value */
   u_char  *statP,         /* pointer returned by var_usmUser */
   int      statLen)       /* *var_len" from var_usmUser */
{
   int rcvd_value ;

   USE_ARG(var_val_len);

   /* Get the value received in SNMP-Set message */
   setVariable(var_val, var_val_type, (u_char *)&rcvd_value, statLen);

   if ( rcvd_value == (int) *statP )
   {
      /* That's correct. The value received is same as the one which we sent */
      return 0;
   }
   else
   {
      /* Oops looks like a conflict. The values are different */
      return -1;
   }
}
```

The above code was referenced from the file `v3mib.c`.

6 SNMPv3 Design Details

6.1 Dynamic Memory Allocation

As a general note, all allocations are done using `V3_ALLOC()` and de-allocations are done using `V3_FREE()`. Handling of packets received from the lower layer, or to be sent to the lower layer is an exception for which the functions `v3_pkt_alloc()` and `v3_pkt_free()` are used. They use specific methods for each of the supported transports. For example, for SNMPv3 over UDP, `v3_pkt_alloc()` would use `v3_udp_alloc()` and `v3_pkt_free()` would use `v3_udp_free()`.

6.2 SNMP Packets

Definition of V3_VIRTUAL_PKT

As we are using zero-copy mechanisms, we do not allocate a packet for SNMP. Instead we use the PACKET already passed by the lower, UDP, layer. At the same time, we want to make the SNMPv3 implementation independent of the packet formats used by the lower layer. To accomplish this, the concept of `V3_VIRTUAL_PKT` is used. When we receive a packet from the lower layer, we form a `V3_VIRTUAL_PKT` and then use it. It has pointers to the actual packet. Similarly, when we need to send a SNMPv3 packet, we form a `V3_VIRTUAL_PKT` and then use the underlying transport to send it. This following example struct makes the most of the SNMPv3 code independent of InterNiche's `PACKET` structure.

```
struct v3_virtual_pkt
{
   uchar *buf;              /* Pointer to buffer/SNMP_DATA_AREA */
                            /* Modified by parse and build routines. */
   unsigned len;            /* length */
                            /* Modified by parse and build routines. */
                            /* For a received pkt, value in len field informs about
                             * the number of bytes still to be processed .
                             * For a to_be_sent pkt, len field informs about length
                             * of pkt at a particular stage */
   uchar *pkt;              /* This is used for compatibility with local stacks.
                             * Used only by v3_udp_alloc(), v3_udp_free(), and
                             * v3_udp_send(). Say for InterNiche's IP stack , it
                             * would contain a pointer to PACKET structure */
   uchar *orig_buf;         /* Pointer to start of received pkt. It is never
                             * changed during parsing or building a pkt. */
   unsigned orig_len;       /* Length of the received pkt. It is never
                             * changed during parsing or building a pkt. */
   struct GlobalPara   gp ; /* global-para                    */
   struct SecurityPara sp ; /* usec-para                      */
   struct ContextPara  cp ; /* context-para : engine id, name */
   struct PduPara      pp ; /* pdu-para minus varbinds        */
   struct v3_host      td ; /* transport-domain               */
};
typedef struct v3_virtual_pkt * V3_VIRTUAL_PKT ;
```

This concept is similar to the RIP_VIRTUAL_PKT concept used in InterNiche RIP. There is one basic difference between a RIP packet and an SNMP packet. That is of BER encoding. All SNMP packets (on the wire) are BER encoded, while RIP has direct fixed size fields in it. So while in RIP we could directly map "structs" onto the received packet buffer, we can't do that for SNMP. Hence we have extra variables to store the fields of the SNMP packet. We have representations for all the fields except the variable-bindings of the PDU.

Memory Allocations Related to SNMP Packets

When a PACKET is received by the lower layer, it calls a callback (for example: v3_udp_recv()) for SNMPv3 processing. The following methodology is used regarding memory allocation.

1. It is SNMPv3 module's responsibility to free the received packet. It will free the packet using v3_pkt_free().
2. To do SNMPv3 processing on the received packet, a V3_VIRTUAL_PKT is allocated using the V3_ALLOC() function.
3. When SNMPv3 processing is done on the received packet, first the packet is freed using v3_pkt_free() and then the V3_VIRTUAL_PKT is freed using V3_FREE function.

When a PACKET is to be sent to the lower layer, the following methodology is used.

1. V3_VIRTUAL_PKT is allocated using V3_ALLOC().
2. V3_VIRTUAL_PKT points to a real packet. So, the real packet is allocated using v3_pkt_alloc().

3. Data is populated in the new packet.
4. SNMPv3 packet is sent using `v3_pkt_send()`. `v3_pkt_send()` would use the underlying transport. For example, for UDP, it would use `v3_udp_send()` to send the packet. InterNiche's UDP layer frees the packet after sending. So we should not call `v3_pkt_free()` for a successfully sent packet on InterNiche's UDP layer.
5. After the packet is sent, `V3_FREE()` is called to free the `V3_VIRTUAL_PKT`. As discussed in the previous step, the packet allocated with `v3_pkt_free()` will be freed by the lower layers. So we just need to free the virtual packet which was allocated using `V3_ALLOC()`.

6.3 Implementation of Tables

Generic Implementation: Basic Structures

genlist.c has a generic implementation of a list of elements. The list has a name, and addition, deletions, lookups in the list can be done. This implementation can be used to implement multiple lists, each list having its own entries. For SNMPv3, each table is implemented as a list. Each row of the table is a single element in the list. So rows can be added, deleted in the tables. Each row should have the first element as an id (identifier), and second element as a name. After that, any table specific data can follow. Basic indexing in the list is done using either the id, or the name, or both. For now, let us assume that this specification is sufficient to implement all tables. We will explain the special case of multiple indices later.

We define the basic structures of a list and then how to use them. All elements are stored in a linked list, and hence NicheList just needs to remember the first element. As the element that it represents can vary in size, it remembers the length. When a new element is to be added to a list, a V3_ALLOC (len_of_element) is done. When an element has to be deleted, V3_FREE() is done.

```
struct NicheList
{
   NICHE_ELE head ;          /* First element of the list */
   int       len_of_element; /* Len of the struct representing element/data*/
};
typedef struct NicheList * NICHELIST;
```

NICHE_ELE is a pointer to an element in the list. Following is its definition. It has a pointer to data (GEN_STRUCT), and a pointer to next element.

```
struct NicheElement
{
   GEN_STRUCT p_data;              /* Pointer to element/data */
   struct     NicheElement *next;  /* Pointer to next data element */
};

typedef struct NicheElement * NICHE_ELE;
```

The following is the definition of GEN_STRUCT. It has two members defined as id and name [MAX_NAME_LEN]. The list can be a singly indexed where all elements are uniquely identified by either id or name. Or it can be doubly indexed where all elements are uniquely identified by the id/name combination.

```
struct TemplateStruct
{
   long id;
   char name[MAX_NAME_LEN];
};

typedef struct TemplateStruct * GEN_STRUCT ;
```

Next, we have the functions which use the above structures and provide functionality for basic operations.

```
int         niche_list_constructor  (NICHELIST  list,int len_of_ele);
int         niche_list_destructor   (NICHELIST  list);
int         niche_add               (NICHELIST  list,GEN_STRUCT ptr_data);
int         niche_add_id_and_name   (NICHELIST  list,long id,char *name);
int         niche_del               (NICHELIST  list,GEN_STRUCT ptr_data);
int         niche_del_id            (NICHELIST  list,long id);
int         niche_del_name          (NICHELIST  list,char *name);
int         niche_del_id_and_name   (NICHELIST  list,long id,char *name);
GEN_STRUCT  niche_lookup_id         (NICHELIST  list,long id);
GEN_STRUCT  niche_lookup_name       (NICHELIST  list,char *name);
GEN_STRUCT  niche_lookup_id_and_name(NICHELIST  list,long id,char *name);
int         niche_lookup_multi_match(NICHELIST  list,long id,char *name,
                                          GEN_STRUCT matches[ ]);
int         niche_list_show         (NICHELIST  list);
int         niche_list_len          (NICHELIST  list);
GEN_STRUCT  niche_list_getat        (NICHELIST  list,int index);

int         niche_element_show      (GEN_STRUCT ptr_data);
```

Generic Implementation : Usage

The implementation of GEN_STRUCT has two fields, number and name. In places where only one of them is used for indexing, the other field can be used to store some other value or be left blank. If the name field is not used, then the size of char array can be reduced to 1 char to save memory. GEN_STRUCT is perfect for the following implementations.

1. A list having number based indexing
2. A list having name based indexing
3. A doubly indexed list based on number and name

In all the above cases, it is assumed that all entries can be uniquely identified based on the index. All the functions for list manipulation can be used as such. The function niche_lookup_multi_match() has no significance in this context and should not be used.

For a list having more than two indices, the addition and deletion functions remain the same. The LOOKUP problem in this case is that the number, name based dual index can't identify a unique entry. Hence a special function has been provided. It is niche_lookup_multi_match(). It is an extension of the function niche_lookup_id_and_name(). It accepts one more argument, which is a array of pointers. All the matched entries are returned via this array. This function returns the "number of entries in the array".

The following steps illustrate a sample usage of Generic List.

1. Call niche_list_constructor() to initialize a new list.

```
struct AppElement
{
   long id;
   char   name[MAX_NAME_LEN];
};

struct NicheList app_list;
NICHELIST p_app_list = &app_list;

niche_list_constructor(p_app_list,sizeof(AppElement));
```

2. Call niche_add*() to add elements to the list.

```
struct AppElement ele1 = { 1000, "router" };
niche_add(p_app_list,&ele1);
niche_add_id_and_name(p_app_list,1000,"bridge");
niche_add_id_and_name(p_app_list,1000,"gateway");
niche_add_id_and_name(p_app_list,1001,"gateway");
```

3. Use `niche_lookup*()` to find elements in the list.

```
struct AppElement *ele2, *ele3;
ele2=niche_lookup_id(p_app_list,1000);  /* Looks for first match */
if ( ele2 == NULL )
{
   // not found
}
ele3=niche_lookup_id_and_name(p_app_list,1000,"router");
if ( ele3 == NULL )
{
   // not found
}
```

4. Use `niche_del*()` to delete elements from the list.

```
niche_del_id(p_app_list,1000);  // Delete all elements with "id=1000"
niche_del_id_and_name(p_app_list1001,"gateway");
```

5. Use `niche_list_destructor()` delete all elements from the list.

```
niche_list_destructor(p_app_list);
```

Implementing a Table Using the Generic Implementation

We will illustrate how the `GroupTable` has been implemented in SNMPv3. Let us first define what goes in the header file.

```
/* the LIST that would hold all the groups */
extern NICHELIST p_group_list;

struct GroupTable
{
   u_long sec_model                  ;
   char   sec_name     [SNMP_NAME_LEN];
   char   group_name   [SNMP_NAME_LEN];
   int    storage_type               ;
   int    status                     ;
};

typedef struct GroupTable * GRP_TABLE;

int    v3_group_tbl_init   ();
int    v3_group_tbl_cleanup();
```

`p_group_list` represents the Group Table. `GroupTable` represents the content of each row of this table. Following is a description of the functions:

- `v3_group_tbl_init()` - Initialize the group table and add some static rows to it. This function should be called in `v3_init()`.
- `v3_group_tbl_cleanup()` - Deallocate all the memory that was allocated for the group table. This function should be called in `v3_cleanup()`.

```
int
v3_add_group(void *pio)
{
   return v3_add_entry(pio,v3_get_sec_num(GROUP_SECTION));
}
int
v3_del_group(void *pio)
{
   return v3_del_entry(pio,v3_get_sec_num(GROUP_SECTION));
}
```

With the above implementation in place, the `niche_*` functions, `niche_add()`, `niche_del()`, and `niche_lookup_name()` mentioned earlier, can be used to perform various operations on this table.

6.4 Dependency on SNMP Core Implementation

MIB Instrumentation

SNMPv3 uses the MIB instrumentation of SNMP core. Please refer to the SNMPv1/V2c Reference Manual for learning how to integrate MIBs with the SNMP module.

To do MIB related operations, SNMPv3 uses the following functions of SNMP core.

`getStatPtr()`	Search for an `OID` in the `OID` list.
`goodValue()`	Check whether the value to be set is a good value
`setVariable()`	Set the value of an `OID`
`compare()`	Compare two `OID`s
`set_parms()`	Initialized before calling `setVariable()`

BER Encoding

To parse SNMPv3 packets and to build SNMPv3 packets, SNMPv3 uses the basic BER encoding related functions of SNMP core. Following are functions used:

- `asn_parse_header()`
- `asn_parse_header2()`
- `asn_parse_string()`
- `asn_parse_int()`
- `snmp_parse_var_op()`
- `asn_build_int()`
- `asn_build_string()`
- `asn_build_header()`
- `snmp_build_var_op()`

6.5 Response to Discover Packets

A SNMPv3 manager needs to know the `EngineId` of the SNMPv3 agent. Only then can it manage the agent. To find out the `EngineId` of an agent, the manager sends a discover packet. The agent's response to this discover packet contains the agent's `EngineId`.

According to RFC2574, "*The response to Discovery message will be a Report message containing the snmpEngineID of the authoritative SNMP engine as the value of the msgAuthoritativeEngineID field within the msgSecurityParameters field. It contains a Report PDU with the usmStatsUnknownEngineIDs counter in the varBindList.*"

InterNiche SNMPv3 agent works in accordance with the RFC specifications.

7 Port Provided Functions

The functions described in this section must be provided by the porting programmer as part of the porting the InterNiche SNMPv3. If you are using the InterNiche TCP/IP stack, many for these functions are already provided.

In InterNiche reference ports, these functions are either mapped directly to system calls via macros in `v3port.h`, or they are implemented directly in `v3port.c`.

7.1 General Functions

dprintf

Name

`dprintfinfo_printf()`

Syntax

```
void dprintf(char *, ...);void info_printf(char *,...);
```

Description

These routines are functionally the same as printf. They are called by the stack code to inform the programmer or end user of system status. `dprintf()` prints error warnings during runtime and `info_printf()` is used to display informational messages.

For example, `dprintf()` would be used to display errors and `info_printf()` to display information about processing that happens in the background (such as the arrival of a packet, etc.).

dtrap

Name

`dtrap`

Syntax

```
void dtrap(void);
```

Description

This primitive is intended to hook a debugger whenever it is called. It aids in debugging.

Returns

Usually nothing, depending on user modifications.

ENTER_CRIT_SECTION

Name

`ENTER_CRIT_SECTION()EXIT_CRIT_SECTION()`

Syntax

```
void ENTER_CRIT_SECTION (void);void EXIT_CRIT_SECTION (void);
```

Description

These two primitives should be designed to be paired around sections of code that must not be interrupted or pre-empted. Generally these simply need to disable and re-enable interrupts. On UNIX-like systems they can be mapped to the `spl()` primitive. On Windows DLLs they can be defined to `NULL` functions since Windows message based system always runs to completion. Examples for various operating environments are available from InterNiche upon request. Only the definitions are given here; for examples see the source code.

The stack source code always pairs these two in the same routines, the implementers can push values on the stack in `ENTER` and retrieve it in the following `EXIT`. The Intel x86 example takes advantage of this to push the existing flags register on the stack, saving the interrupt flag state, and retrieves the value for the flags resister later, restoring the interrupt flag as it was before the `ENTER` call.

7.2 SNMPv3 Transport Layer Interface

v3_pkt_alloc

Name

`v3_pkt_alloc`

Syntax

```
V3_VIRTUAL_PKT v3_pkt_alloc(int size,V3_VIRTUAL_PKT p_v3pkt, int flags);
```

Parameters

Size of memory block (IN)
Pointer to V3_VIRTUAL_PKT (OUT)
Flags informing about the underlying transport

Description

This function allocates memory for a SNMPv3 packet and forms a `V3_VIRTUAL_PKT` with it. `V3_VIRTUAL_PKT` is used to store the buffer and length for the packet. It is assumed that the second argument (pointer to `V3_VIRTUAL_PKT`) points to a valid structure (already allocated structure).

The `flags` field informs about the underlying transport. If the `HF_IPADDR` flag is set, then the underlying transport is TCP/IP and `v3_udp_alloc` is called.

Returns

Returns pointer to `V3_VIRTUAL_PKT` or `NULL`.

v3_pkt_free

Name

`v3_pkt_free`

Syntax

```
int v3_pkt_free(V3_VIRTUAL_PKT p_v3pkt);
```

Parameters

Pointer to V3_VIRTUAL_PKT (OUT)

Description

This function frees the memory that was allocated using `v3_pkt_alloc()`. The argument to this function is a pointer to `V3_VIRTUAL_PKT` which was populated using `v3_pkt_alloc()`.

The `flags` field of `p_v3pkt` informs about the underlying transport. If the `HF_IPADDR` flag is set, then the underlying transport is TCP/IP and `v3_udp_free()` is called.

Returns

Returns SUCCESS or an error number.

v3_pkt_send

Name

`v3_pkt_send`

Syntax

```
int v3_pkt_send(V3_VIRTUAL_PKT p_v3pkt);
```

Parameters

Pointer to V3_VIRTUAL_PKT (packet to be sent) (IN)

Description

This function sends a SNMPv3 packet on the network.

The `flags` field of `p_v3pkt` informs about the underlying transport. If the `HF_IPADDR` flag is set, then the underlying transport is TCP/IP and `v3_udp_send()` is called.

Returns

Returns `SUCCESS` (0) if everything went OK, else returns a non-zero error code.

v3_udp_init

Name

`v3_udp_init`

Syntax

```
int v3_udp_init();
```

Description

This call does the initialization so that SNMPv3 packets can be received via `v3_udp_recv()`.

Returns

Returns SUCCESS (0) if everything went OK, else returns a non-zero error code.

v3_udp_recv

Name

`v3_udp_recv`

Syntax

```
int v3_udp_recv(PACKET p, void *data, struct sockaddr *src);
```

Parameters

p	pointer to the received packet
`data`	callback data (as registered with UDP via `udp_open()`, currently always NULL.
`src`	pointer to `sockaddr_in` (or `sockaddr_i6`) structure containing information about the sender of the SNMP/UDP/IP datagram.

Description

`v3_udp_recv()` is a callback function. It uses parameters from the received packet to make a call to `v3_process_rcvd_pkt()`. The parameters are pointers to the received packet and the source port number of the received packet.

The function as described here is used with the InterNiche TCP/IP stack. The structure PACKET contains the data that is needed to call `v3_process_rcvd_pkt()`.

v3_udp_send

Name

`v3_udp_send`

Syntax

```
int v3_udp_send(u_short dst_port, u_short src_port, V3_VIRTUAL_PKT p_v3pkt);
```

Parameters

`dst_port`	Destination port number (IN)
`src_port`	Source port number (IN)
`p_v3pkt`	Packet to be sent (IN)

Description

This function sends a SNMPv3 packet on the network.

Returns

Returns SUCCESS (0) if everything went OK, else returns a non-zero error code.

v3_udp_cleanup

Name

`v3_udp_cleanup`

Syntax

```
int v3_udp_cleanup();
```

Description

This call cleans up the data structures allocated in `v3_udp_init()`.

Returns

Returns SUCCESS (0) if everything went OK, else returns a non-zero error code.

v3_udp_alloc

Name

`v3_udp_alloc`

Syntax

```
V3_VIRTUAL_PKT v3_udp_alloc(int size,V3_VIRTUAL_PKT p_v3pkt);
```

Parameters

`size`	size of memory block (IN)
`p_v3pkt`	Pointer to V3_VIRTUAL_PKT (OUT)

Description

This function allocates memory for a SNMPv3 packet and forms a `V3_VIRTUAL_PKT` with it. `V3_VIRTUAL_PKT` is used to store the buffer and length for the packet. It is assumed that the second argument (pointer to `V3_VIRTUAL_PKT`) points to a valid structure (already allocated structure).

Returns

Returns pointer to V3_VIRTUAL_PKT or NULL.

v3_udp_free

Name

`v3_udp_free`

Syntax

```
int v3_udp_free(V3_VIRTUAL_PKT p_v3pkt);
```

Parameters

Pointer to `V3_VIRTUAL_PKT` (OUT)

Description

This function frees the memory that was allocated using `v3_udp_alloc()`. The argument to this function is a pointer to `V3_VIRTUAL_PKT` which was populated using `v3_udp_alloc()`.

Returns

Returns SUCCESS or error number.

7.3 Timer Support

SNMPv3 checks for timeliness, and hence needs some timer related support from the OS. Mainly a means to know current clock tick, and number of ticks per second.

Name

`v3_start_timer()v3_check_timeout()v3_get_seconds_since_up()`

Syntax

```
void v3_start_timer(unsigned long *timer);int v3_check_timeout(unsigned long timer,u_long interval);u_long v3_get_seconds_since_up(void);
```

Parameters

`timer`	represents the value in clock-ticks since the system was up
`interval`	represents the time interval in seconds

Description

These functions are used to implement a timer. When a timer is to be started, `v3_start_timer()` is called. Then, `v3_check_timeout()` is called periodically (once every clock-tick) to find out if a timeout has occurred.

`v3_get_seconds_since_up()` is used to find out the number of seconds since the SNMpv3 engine is up.

Returns

`v3_start_timer()`	Nothing
`v3_check_timeout()`	Returns SUCCESS (0) if timeout has occurred, else FAILURE.
`v3_get_seconds_since_up()`	returns the number of seconds.

7.4 The SNMPv3 Entry Points

When SNMPv3 is ported to any transport later, the SNMPv3 entry points are the only things that the external module(s) need to worry about. These entry points are the interface between SNMP and the other modules.

Another function that can be the entry point between SNMPv3 and other modules is `v3_udp_recv()`. For example, in the InterNiche SNMPv3 implementation, this function is used as an callback when a packet arrives on any SNMP port.

Name

`v3_init()v3_check ()v3_cleanup()`

Syntax

```
int v3_init(void);void v3_cleanup(void);void v3_check(void);
```

Description

These functions are the interface between SNMPv3 and other modules. The function `v3_init()` is the initialization function and should be called after the transport layer, TCP/IP, has been initialized. It initializes the globals used by SNMPv3, initializes and populates various tables used by SNMPv3, and pre-calculates the authentication and privacy keys. Calculating a key from a password takes few seconds, so we don't want to be doing it when the packet arrives. Hence the keys are pre-calculated at initialization time.

The function `v3_check()` doesn't have much functionality. It does the USM (User-based Security Model) timeliness.

Returns

`v3_check()`	Nothing
`v3_init()`	SUCCESS (0) or an error code
`v3_cleanup()`	SUCCESS (0) or an error code

8 SNMPv3 Table Manipulation Functions

SNMPv3 implements all tables using a Generic List. This section describes all the functions that can be used for table manipulation.

Name

`niche_list_constructor()`

Syntax

```
int niche_list_constructor (NICHELIST list, int len_of_ele);
```

Parameters

list to be initialized (IN/OUT)
Size of each row/entry in the list

Description

This function should be called to initialize a list. It initializes a list, so that all other operations on the list can be performed.

Returns

Returns SUCCESS (0) if everything went OK, else returns a non-zero error code.

Name

`niche_list_destructor()`

Syntax

```
int niche_list_destructor (NICHELIST list);
```

parameters

list to be cleaned up (IN/OUT)

Description

This function should be called to free all the memory used by a list.

Returns

Returns SUCCESS (0) if everything went OK, else returns a non-zero error code.

Name

`niche_add()niche_add_id_name()`

Syntax

```
int niche_add (NICHELIST list,GEN_STRUCT ptr_data);int
niche_add_id_and_name (NICHELIST list,long id,char *name);
```

Parameters

`list`	list to be used (IN/OUT)
`ptr_data`	points to data for an entry in the list
`id`	value for an entry in the list
`name`	value for an entry in the list

Description

These functions should be called to add entries to the list. If the entry in the list has more than two members, then it is advised to use niche_add(). If the entry in the list has only two members, namely an id and a name , then niche_add_id_and_name() can also be used to add an entry.

Returns

Returns SUCCESS (0) if everything went OK, else returns a non-zero error code.

Name

`niche_del()niche_del_id()niche_del_name()niche_del_id_and_name()`

Syntax

```
int niche_del (NICHELIST list,GEN_STRUCT ptr_data);int niche_del_id
(NICHELIST list,long id);int niche_del_name (NICHELIST list,char *name);int
niche_del_id_and_name (NICHELIST list,long id,char *name);
```

Parameters

`list`	list to be used (IN/OUT)
`ptr_data`	points to data for an entry in the list
`id`	value for an entry in the list
`name`	value for an entry in the list

Description

These functions should be called to delete entries in the list. niche_del() can be used to delete a particular entry. niche_del_id() can be used to delete all entries matching the value of id. niche_del_name() can be used to delete all entries matching the value of name. niche_del_id_and_name() can be used to delete all entries matching the value of id and name.

Returns

Returns SUCCESS (0) if everything went OK, else returns a non-zero error code.

Name

`niche_lookup_id(), niche_lookup_name()niche_lookup_id_and_name(), niche_lookup_multi_match()`

Syntax

```
GEN_STRUCT niche_lookup_id (NICHELIST list,long id);GEN_STRUCT
niche_lookup_name (NICHELIST list,char *name);GEN_STRUCT
niche_lookup_id_and_name(NICHELIST list,long id,char *name);int
niche_lookup_multi_match(NICHELIST list,long id,char *name, GEN_STRUCT
matches[ ]);
```

Parameters

`list`	list to be used (IN/OUT)
`matches`	array which can hold multiple entries (OUT)
`id`	value for an entry in the list
`name`	value for an entry in the list

Description

These functions should be called to lookup entries to the list `niche_lookup_id()` can be used to lookup the first entry matching the value of id. `niche_lookup_name()` can be used to lookup the first entry matching the value of name. `niche_del_id_and_name()` can be used to lookup the first entry matching the value of id and name.

`niche_lookup_multi_match()` should be used for lists having more than two indices. In this case, a combination of id, name would not identify a unique entry in the list but can identify multiple entries in the list. All the matching entries are returned via the matches array and this function returns the number of matched entries.

Returns

`niche_lookup_id()`	returns the matching entry in the list or `NULL` if no entries are matching
`niche_lookup_name()`	returns the matching entry in the list or `NULL` if no entries are matching
`niche_lookup_id_and_name()`	returns the matching entry in the list or `NULL` if no entries are matching
`niche_lookup_multi_match()`	returns the number of matched entries.

Name

`niche_list_show()niche_list_len()niche_list_getat()niche_element_show()`

Syntax

```
int niche_list_show (NICHELIST list);int niche_list_len (NICHELIST list);
GEN_STRUCT niche_list_getat (NICHELIST list,int index);int
niche_element_show (GEN_STRUCT ptr_data);
```

Parameters

`list`	list to be used (IN/OUT)
`ptr_data`	points to data for an entry in the list
`index`	the numerical value of an element from the start of the list

Description

These functions are used for general information about the list. `niche_list_len()` returns the number of elements in the list. `niche_list_getat()` returns the entry in the list at position index. `niche_list_show()` displays values (id, name) for all entries in the list. It uses `niche_element_show()` to display value for a particular entry.

If the 5th entry is to be deleted from the list, then `niche_list_getat()` can be used to get the 5th entry and then `niche_del()` can be used to delete it.

If all entries in a list are to be processed, then `niche_list_len()` can be used to get the length of the list. Then `niche_list_getat()` can be used to get the entry at each position. This entry can then be processed.

Returns

`niche_list_show()`	returns SUCCESS or error code.
`niche_element_show()`	returns SUCCESS or error code.
`niche_list_len`	returns the length of the list.
`niche_list_getat()`	returns the pointer to an entry in the list.

9 APPENDIX A: snmpv3.script

Here is a sample listing of snmpv3.script that is used to configure the SNMPv3 module. These commands are used when V3_STATIC_TABLES is not enabled in the build.

```
##SNMPv3 script

##add authentication and privacy algorithms
snmpv3 authoid -n NoAuth         -o 1.3.6.1.6.3.10.1.1.1
snmpv3 authoid -n MD5            -o 1.3.6.1.6.3.10.1.1.2
snmpv3 authoid -n SHA1           -o 1.3.6.1.6.3.10.1.1.3
snmpv3 authoid -n NoPriv         -o 1.3.6.1.6.3.10.1.2.1
snmpv3 authoid -n DES            -o 1.3.6.1.6.3.10.1.2.2
snmpv3 authoid -n CFB128-AES-128 -o 1.3.6.1.6.3.10.1.2.4

##add users
snmpv3 username -u SimpleUser   -v SimpleUser   -a NoAuth -b ""               -p NoPriv
-q ""
snmpv3 username -u AuthOnlyUser -v AuthOnlyUser -a MD5    -b AuthOnlyUserAPwd -p NoPriv
-q ""
snmpv3 username -u SHA1OnlyUser -v SHA1OnlyUser -a SHA1   -b SHA1OnlyUserAPwd -p NoPriv
-q ""
snmpv3 username -u MD5User1     -v MD5User1     -a MD5    -b MD5User1APwd     -p DES
-q MD5User1PPwd
snmpv3 username -u MD5User2     -v MD5User2     -a MD5    -b MD5User2APwd     -p DES
-q MD5User2PPwd
snmpv3 username -u AMUser7      -v AMUser7      -a MD5    -b AMUser7MD5APwd   -p CFB128-AES-128
-q AMUser7AESPPwd
snmpv3 username -u DSUser7      -v DSUser7      -a SHA1   -b DSUser7SHA1APwd  -p DES
-q DSUser7DESPPwd
snmpv3 username -u ASUser7      -v ASUser7      -a SHA1   -b ASUser7SHA1APwd  -p CFB128-AES-128
-q ASUser7AESPPwd

##add groups
snmpv3 group -g initial -n SimpleUser   -m 3
snmpv3 group -g initial -n AuthOnlyUser -m 3
snmpv3 group -g initial -n SHA1OnlyUser -m 3
snmpv3 group -g initial -n MD5User1     -m 3
snmpv3 group -g initial -n MD5User2     -m 3
snmpv3 group -g initial -n AMUser7      -m 3
snmpv3 group -g initial -n DSUser7      -m 3
snmpv3 group -g initial -n ASUser7      -m 3

##add views
snmpv3 mibview -n internet   -o 1.3.6.1            -m ""
snmpv3 mibview -n restricted -o 1.3.6.1.2.1.1      -m ""
snmpv3 mibview -n restricted -o 1.3.6.1.2.1.11     -m ""
snmpv3 mibview -n restricted -o 1.3.6.1.6.3.10.2.1 -m ""
snmpv3 mibview -n restricted -o 1.3.6.1.6.3.11.2.1 -m ""
snmpv3 mibview -n restricted -o 1.3.6.1.6.3.15.1.1 -m ""

##add access table
```

```
snmpv3 access -m 3 -g initial -l 0 -c DefaultContextName -x -r restricted -w restricted -n
restricted
snmpv3 access -m 3 -g initial -l 1 -c DefaultContextName -x -r internet   -w internet   -n
internet
snmpv3 access -m 3 -g initial -l 3 -c DefaultContextName -x -r internet   -w internet   -n
internet

##add trap address table
snmpv3 taddr -n mongo -a 10.0.0.76 -v monitor -r 5 -x 60 -z mongo
snmpv3 taddr -n stein -a 10.0.0.144 -v monitor -r 5 -x 60 -z stein
snmpv3 taddr -n mongov6 -a fe80::240:f4ff:feed:8b77%2 -v monitor -r 5 -x 60 -z mongo

##add trap parameters table
snmpv3 tparam -m 3 -n mongo -u 3 -v AuthOnlyUser -l 1
snmpv3 tparam -m 3 -n stein -u 3 -v ASUser7 -l 3
```

10 Glossary

authentication

The process of ensuring message integrity and protection against message replays. It includes both data integrity and data origin authentication.

authoritative SNMP engine

One of the SNMP copies involved in network communication designated to be the allowed SNMP engine to protect against message replay, delay, and redirection. The security keys used for authenticating and encrypting SNMPv3 packets are generated as a function of the authoritative SNMP engine's engine ID and user passwords. When an SNMP message expects a response (for example, get exact, get next, set request), the receiver of these messages is authoritative. When an SNMP message does not expect a response, the sender is authoritative.

community string

A text string used to authenticate messages between a management station and an SNMP v1/v2c engine.

data integrity

A condition or state of data in which a message packet has not been altered or destroyed in an unauthorized manner.

data origin authentication

The ability to verify the identity of a user on whose behalf the message is supposedly sent. This ability protects users against both message capture and replay by a different SNMP engine, and against packets received or sent to a particular user that use an incorrect password or security level.

encryption

A method of hiding data from an unauthorized user by scrambling the contents of an SNMP packet.

group

A set of users belonging to a particular security model. A group defines the access rights for all the users belonging to it. Access rights define what SNMP objects can be read, written to, or created. In addition, the group defines what notifications a user is allowed to receive.

notification host

An SNMP entity to which notifications (traps and informs) are to be sent.

notify view

A view name (not to exceed 64 characters) for each group that defines the list of notifications that can be sent to each user in the group.

privacy

An encrypted state of the contents of an SNMP packet where they are prevented from being disclosed on a network. Encryption is performed with an algorithm called DES-CBC (Cipher Block Chaining - Data Encryption Standard) or CFB128-AES-128 (Advanced Encryption Standard in Cipher Feedback Mode).

read view

A view name (not to exceed 64 characters) for each group that defines the list of object identifiers (OIDs) that are accessible for reading by users belonging to the group.

security level

The three security levels are:

- noAuthNoPriv (neither authentication nor privacy)
- authNoPriv (authentication only, no privacy)
- authPriv (authentication and privacy)

security model

The security strategy used by the SNMP agent. InterNiche supports three security models: SNMPv1, SNMPv2c, and SNMPv3.

Simple Network Management Protocol (SNMP)

A network management protocol that provides a means to monitor and control network devices, and to manage configurations, statistics collection, performance, and security.

Simple Network Management Protocol Version 2c (SNMPv2c)

The second version of SNMP, it supports centralized and distributed network management strategies, and includes improvements in the Structure of Management Information (SMI), protocol operations, management architecture, and security.

SNMP engine

A copy of SNMP that can either reside on the local or remote device.

SNMP group

A collection of SNMP users that belong to a common SNMP list that defines an access policy, in which object identification numbers (OIDs) are both read-accessible and write-accessible. Users belonging to a particular SNMP group inherit all of these attributes defined by the group.

SNMP user

A person for which an SNMP management operation is performed. For informs, the user is the person on a remote SNMP engine who receives the informs.

SNMP view

A mapping between SNMP objects and the access rights available for those objects. An object can have different access rights in each view. Access rights indicate whether the object is accessible by either a community string or a user.

write view

A view name (not to exceed 64 characters) for each group that defines the list of object identifiers (OIDs) that are able to be created or modified by users of the group.