SNMPv1-v2c Agents and MIB Compiler Technical Reference

Interniche Legacy Document

Version 1.00

Date: 18-May-2017 12:32

Table of Contents

1 Introduction

This Technical reference is provided with the InterNiche Portable SNMP protocol stack sources. The purpose of this document is to provide enough information so that a moderately experienced "C" programmer with a reasonable understanding of SNMP can port the InterNiche SNMP to a new environment. Experience with networking code, especially TCP/IP, is helpful to understanding many of the concepts herein.

This manual can be used for all SNMP agents, namely SNMPv1, SNMPv2c and SNMPv3. The MIB instrumentation is common for all agents. If any section refers only to a specific SNMP version, then it is explicitly mentioned. The porting effort required for SNMPv1 or SNMPv2 is the same and hence the same manual can be used. For SNMPv3, additional porting effort is needed and is described in a separate document.

1.1 Reference Implementation

It is also assumed that the InterNiche "reference port" sources are available as a reference.

If you ordered SNMP along with the InterNiche TCP/IP stack, the source code you received should be ready to build an executable SNMP which implements the MIB-II (RFC1213) agent. You should be able to use this as a reference and add support of other MIBs of your interest. **It is important to note** that as delivered, only a small number of RFC1213 objects are available. To enable all objects, please disable `SNMP_MIN_BUILD` in `h/snmpport.h`.

1.2 Terms and Conventions

In this document, the term "agent", when used without other qualification, means the InterNiche SNMP agent 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. The agent is delivered with example implementation notes for Sockets as many embedded systems already have sockets. A copy of the Sockets API documentation is available from InterNiche upon request. A "user" or "porting engineer" usually refers to the engineer who is porting the server. An "end user" refers to the person at the management station who ultimately ends up using the "user's" product.

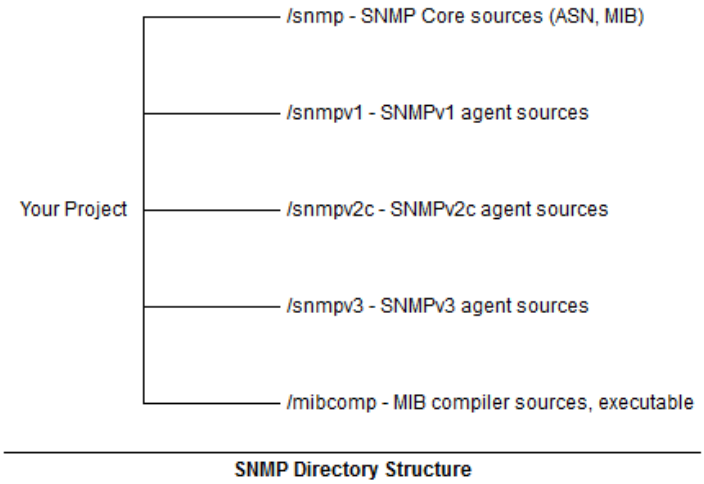
Names of files, C structures, and C routines are displayed in the following format: `c_routine()`.

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

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

1.3 Directory Trees

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. The section starting with SNMP Core Files details the names and functions of the provided files, as well as information on which files should and should not be modified.

1.4 Requirements

Before beginning a port, the programmer should ensure that the necessary resources are available in the target environment. This step is usually done before you purchase the InterNiche package so it is not listed in the steps above.

Obviously, there must be a processor (with some spare CPU power) with some sort of operating system or monitor, some RAM memory, and some sort of network interface. The exact amounts of these resources will vary depending on the type and number of MIB variables, the processor being used, and the responsiveness required from the system.

A Networking stack is also required. InterNiche sells a TCP/IP stack which supports SNMP. If the target platform already has an appropriate protocol stack, then the InterNiche stack will not be needed. InterNiche SNMP has been ported to UNIX Sockets, Windows WinSock, NCSA Telnet, and several proprietary stacks.

The MIB variables to be defined and implemented will drive all the other requirements. In general, the more MIB variables defined, the more memory will be required for both code and data. Variables defined in a table (part of an ASN.1 SEQUENCE) will generally require more code space and more CPU power than non-tabular variables. Code written (or re-written) to maintain the variables in the structures produced by the InterNiche MIB compiler's -h option will also tend to be more efficient than code adapted to an existing system. Systems which maintain the implemented variables in hashed or ordered tables will also tend to be faster than those which don't.

The most reasonable approach to assessing processing power requirements is to consider similar systems and proceed from there. Variables can be added and deleted fairly quickly using the MIB compiler and then tested on various. Processes can be run simultaneously with the agent to determine the effect of heavy SNMP work on the performance of the system.

2 MIB Compiler

The InterNiche MIB compiler is a program which takes as input the SNMP MIB specification written in ASN. 1 compliant notation and produces one or more of these output files:

- C language source code files which are useful as a starting point for implementing SNMP MIBs in SNMP agents. These files contain skeleton routines which will need to be filled in by the porting programmer.
- A variables file which contains a table that links MIB variables to the skeleton routines.
- .h files with the function prototypes and definitions for the above C files.
- A "numbers" file, .num, suitable for describing the MIB variables to an SNMP management station. The format of the numbers file is defined below.

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

2.1 Building the Compiler

The MIB compiler is provided in both source and executable forms for DOS, so if you are developing on a DOS system you can skip this section. For other platforms (i.e. UNIX), you will need to build mibcomp using the system's native C compiler. The sources consist of two `.c` files and one `.h` file. These files are:

parse.c	reads MIB files, outputs the specified output files
parse.h	defines for mibcomp.
free.c	all routines to free various data structures
nextcarg.c	parses a string and returns a pointer to next argument
memwrap.c	memory wrapper code for dynamic memory managment during debugging
memwrap.h	defines API for memwrap.c
tree.c	parses all the nodes, forms a tree, and generates output files

Keep in mind that the MIB compiler will probably be used regularly during the development of the SNMP agent as MIB variables are changed. The mibcomp executable should be stored in a directory where it can be invoked by the makefiles which build your embedded system. On DOS, the compiler is named mibcomp. exe. Most UNIX compilers default to building a file named a.out, and for consistency we recommend you rename it to mibcomp.

2.2 Usage

The MIB compiler's command line format is as follows:

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

Several of these options generate additional information. Please refer to file `parse.h` for more information.

2.3 Input

The MIB Compiler takes as input one or more MIB definition files as described by RFC1155. Several samples of such files are shipped with the InterNiche SNMP package. These sample files were created using a simple text editor to edit out the non-ASN.1 portions for the RFCs each MIB came from. By deleting all the text preceding the `DEFINITIONS ::= BEGIN` line, all the text following the `END` line, and all the page break text (footer, page break, header), one is left with the ASN.1 which is suitable for input to the InterNiche compiler.

It is, of course, possible to define your own MIBs. Deciding which variables you want to manage and how to organize them are by far the hardest part. Writing the ASN.1 text to describe them is relatively easy. Once the MIB has been written in RFC1155 compliant ASN.1, it can be compiled and implemented like any standard MIB.

Input Caveats

- The MIB compiler has the MARCOs found in RFC2578 and RFC2579 built-in :

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

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

- The provided files rfc2578.mib and rfc2579.mib files eliminate these macros and other pre-defined base and built-in ASN.1 types. The compiler does not support redefinition of these macros and built-in types so they must not appear in any `.mib` files.

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

2.4 Output

As mentioned above, the compiler will produce `.c` files, `.h` files, and numbers files. Since these files will be re-generated whenever the MIB compiler is run (i.e. whenever the MIB is changed) you should avoid editing them by hand. The files are:

snmpvars.c	will be produced which contains the table mapping the MIB variables and groups to the stub routines which get and set your variables.
.h (include) file	with prototypes for the C stub routines, token definitions for the MIB variables, and suggested data structures to contain the variables for each group or sequence in the MIB.
.c file	containing stub routines for accessing MIB variables. These are intended to be copied to your own source files and completed there.
.num file	containing all the MIBs' Object Id s in dot notation with the corresponding symbols.

The names of the .c and .h files will be determined by the first 8 characters of the name given on the DEFINITIONS line in the last input MIB file. For example, RFC1213's `DEFINITIONS` line is as follows:

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

This results in a file name of `rfc1213_.c`. The names of the other output files are determined similarly; in this case: `rfc1213_.h` and `rfc1213_.num`.

The `snmpvars.c` file contains the variables structure which the SNMP agent code uses to associate a C routine with individual variables. More on the variables structure in the next section.

The `.c` file contains stubs for the C routines prototyped in the .h file. These routines are framed and commented, however they are only empty frames with no internal code. The areas where code needs to be added to actually implement the variables are flagged with the text TODO (all in CAPS, as shown). Part of the work of implementing a new MIB is to replace the TODO lines with C code which performs the intended `SET` or `GET` operation and returns the correct values. More details on this are given below in the section on C Routine Frames.

C Routine Frames

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

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

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

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

Now that the variables in and out of the C routines have been described, we present the filled in version of var_system() from `snmp/rfc1213.c`

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

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

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

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

Suggested Data Structures

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

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

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

...and the suggested structure.....

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

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

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

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

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

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

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

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

Variables Structure

The discussion of the MIB compiler's output from this point on assumes an understanding of such SNMP concepts as lexicographic ordering, ObjectIds and their components, MIB groups, and ASN.1 SEQUENCEs. Explaining all these is beyond the scope of this chapter. To learn more about them, pick up one of the several good books on SNMP available at computer book stores. A good starting point is The Simple Book by Marshall Rose, published by Prentice-Hall or SNMP, SNMPv2, SNMPv3, and RMON 1 and 2 by William Stallings, published by AddisonWesley.

The InterNiche SNMP agent's internal routines access the MIB variables via a variables table which the MIB compiler produces in the `snmpvars.c` file. The table, named variables, is an array of variable structures. The structure is reproduced below as currently defined. You should always refer to `snmp_var.h` as the final authority.

```
struct variable   {
     oid      name[DEF_VARLEN]; /* object identifier of variable */
     u_char   namelen;          /* length of above */
     char     type;             /* type of variable, INTEGER or (octet) STRING */
     u_char   magic;            /* passed to function as a hint */
     u_short  acl;              /* access control list for variable */
     u_char  * (*findVar)(struct variable *, oid *, int *, int,int *);
   /* function that finds variable */
#ifdef MIB_VALIDATION
     struct size_info  ranges;  /* Set of ranges specified in MIB file */
#endif
};
```

One of these structures is defined for each accessible variable in the input MIBs. The example compiler output from the .c file for the beginning of the MIB-II system group is shown below. Note that if MIB_VALIDATION has been defined there will be additional information in each entry.

```
 struct variable variables[ ] = {
   {{1,3,6,1,2,1,1,1,0},   9, STRING, SYSDESCR, RONLY, var_system },
   {{1,3,6,1,2,1,1,2,0},   9, OBJID, SYSOBJECTID, RONLY, var_system},
   {{1,3,6,1,2,1,1,3,0},   9, TIMETICKS, SYSUPTIME, RONLY, var_system},
   {{1,3,6,1,2,1,1,4,0},   9, STRING, SYSCONTACT, RWRITE, var_system };
}
```

Each entry in the variables table starts with the Object Id of the variable, stored as an array of unsigned values. The next field is the length of the Object Id. In this case they are all nine values, however this length will vary. Next is a token describing the type of the variable. Jumping forward a bit, the final item in the struct is a pointer to a routine used to access the group or sequence of which the variable is member, in this case var_system(), since all variables are members of the system group. var_system() is one of the routines prototyped and stubbed in by the compiler. One of these routine stubs is generated by the compiler for each group or sequence in the input MIBs.

The variable structure's magic field is a token passed to the access routine so it can determine which of the variables in its group or sequence to act on. These magic tokens are also generated by the MIB compiler, and are unique within their group or sequence. They usually start at 0, and then increment throughout the group or sequence by 4s. This is so they can be used to index byte-wise into the suggested Group structures produced for each group or MIB in the .h file. More about this under C Routine Frames below. The next field, acl, controls access to the variable. In the current InterNiche product values are read-only and read-write, however a complex system with multiple levels of privilege could be implemented by modifying the compiler.

The entries in the variables table are ordered lexicographically by Object Id. This is so that the SNMP agent code can do fast lookups on the table to find matching variables or appropriate `GETNEXT` entries.

2.5 Updating MIBs

During the course of most agent implementations the MIB is changed. This is usually because the definitions in the private enterprise MIBs evolve as the project progresses, but industry standard MIBs may also change periodically. When this happens, the existing C code must be edited to reflect the changes.

After editing the C code, you must rerun the MIB compiler to produce new variables, .c and .h files. It is usually a good idea to make the MIB sources (ASN.1 text files) dependencies in your makefile and include a build rule to run the MIB compiler.

If a variable has been deleted, the system will compile and operate as before (except without that variable). It is good form to delete or comment out the stub-derived code which implemented the variable's operations. If the meaning of a variable has been changed, obviously the stub code will have to be changed accordingly.

If new variables have been created, then the new stubs will need to be copied from the output C file and implemented just as the older variables were implemented.

2.6 Validation of Objects as Defined in the MIB

The MIB Compiler can generate validation information for the objects read from the MIB. This information is then used by the SNMP Engine to validate objects at runtime. This feature is optional and can be enabled /disabled with:

```
#define MIB_VALIDATION   1
```

When it is disabled, it reduces the footprint of the SNMP engine.

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

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

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

Some examples of illegal sub-typing (from RFC1902):

`Integer32 (150..100)`	first greater than second
`Integer32 (0..100|50..500)`	ranges overlap
`Integer32 (0|2|0)`	value duplicated
`Integer32 (MIN..-1|1..MAX)`	MIN and MAX not allowed
`Integer32 (SIZE(0..34))`	must not use SIZE
`OCTET STRING (0..100)`	must use SIZE
`OCTET STRING (SIZE(-10..100))`	negative SIZE

The parsed information is stored in the following structure.

```
 #define MCT_MAX_VALUES 10

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

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

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

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

If the porting engineer needs to do something special about validation, this can be done by using a special function that is called during the`SET`operation for the particular object. When such a function is defined for an object, the routine validation is not done even if `MIB_VALIDATION` is enabled. This mechanism is provided so that the porting engineer can override the default method of validation.

To use the validation feature, `MIB_VALIDATION` must be enabled in two places.

1. In `mibcomp/parse.h` enable `MIB_VALIDATION`, recompile the MIB Compiler, and run the MIBs through it. Copy the generated `snmpvars.c` file to the `/snmp` directory. This file has all the validation information for each object.
2. In `snmp/snmp_var.h` enable `MIB_VALIDATION` and recompile the sources in this directory. This will enable the code in the SNMP engine which validates all incoming values of objects during the `SET` operation.

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

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

2.7 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 OIDs in the MIBs.
4. The `snmp-vie.num` file contains a lsit of the MIB objects (from the MIBs that have been compiled), their OIDs and their data types.

We suggest that the files snmpvars.c, snmp-vie.h be placed in the SNMP directory. And that the implementations of functions (prototypes in snmp-vie.c) be placed in implementation directories. For example, lets assume that RFC2578, RFC2579, RFC2571, RFC2572, RFC2573, RFC2574, and RFC2575 are to be added to the default rfc1213.mib and they will be in /snmpv3 directory.

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

The basic idea is as follows:

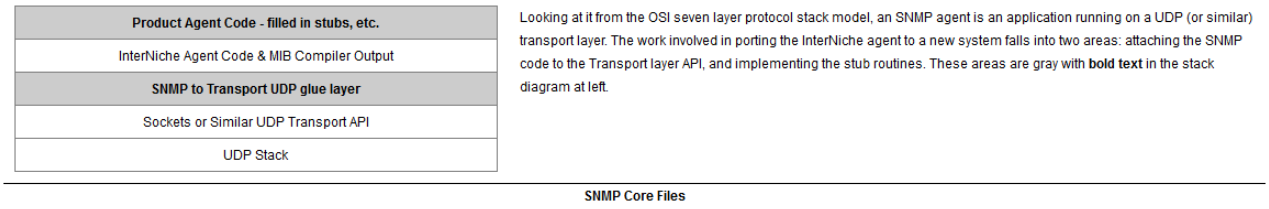
- `snmpvars.c` contains the list of all OIDs supported. All the MIBs are compiled with the "`-vh`" options, and the resulting `snmpvars.c` is placed in SNMP directory. The `.h` file, `snmp_vie.h`, contains declarations for all the prototypes in all the RFCs and is placed in the SNMP directory.
- RFC1213's implementation is shipped with the SNMP agent. Hence we don't want to redo it. To provide MIB instrumentation for the new RFCs, we compile the new RFCs (minus the RFC1213) with the -c option to get a .c file with prototypes. This file is placed in the directory where the MIB instrumentation for new RFCs is to be provided, the SNMPV3 directory.

3 Step by Step Porting Guide for the SNMP Agent

3.1 Getting Started

The ultimate purpose of any SNMP agent is to implement a set of MIB variables which can be read or set by network management applications. InterNiche provides the SNMP layer. The porting programmer must join the SNMP code to a lower transport layer to send and receive messages, and to an upper layer of code which resolves the individual MIB variables into actual values or operations. This is accomplished with the InterNiche tools in the following recommended steps:

1. Determine what MIB variables are to be used. This involves selecting standard MIBs from the RFCs, and possibly writing proprietary MIB extensions. This is usually underway prior to the beginning of the port.
2. Processing the MIBs with the MIB Compiler to create skeletal "stub" C Code files containing variable routines for the port. This is covered in MIB Compiler.
3. Move the SNMP "core" source files(listed below), and the TCP/IP source files if needed, to a directory in the build environment for the target system. Link with the target system and stub routines. Test with a call to snmp_agt_parse().
4. Modify the port files, `snmp/snmpport.c`, `snmpport.h`, `snmpv1/v1port.c` and `snmpv1/v1sock.c` and complete the C code in the stub routines.
5. Compile the new variable routines created in Step 4 and link them with the "core" SNMP files to create an executable image for the new target system.
6. Test and Debug.



SNMP Core Files

The SNMP core source files are listed below. **THESE FILES SHOULD NOT HAVE TO BE MODIFIED FOR A PORT**. If you feel they need to be changed, please discuss it with HCC's technical staff first!

SNMP Directory

- asn1.c
- asn1.h
- npsnmp.h
- parse.h
- snmp.c
- snmp_imp.h
- snmpport.h
- snmputil.c
- snmp_var.h
- specific.c

SNMPv1 Directory

- snmp_age.c
- snmp_aut.c
- trap_in.c
- trap_out.c

SNMPv2c Directory

- v2agent.c
- v2trap.c

Joining the SNMP agent core to non-InterNiche transport layers is accomplished by implementing three simple functions. These are listed below.

```
int   snmp_init( void );
```

`snmp_init()` should be called before any SNMP packets are upcalled. It prepares the SNMP implementation for receiving packets. This can include binding to the Transport layer (e.g. Sockets `listen()` call), preparing an extended community strings array, initializing TRAP targets, etc. It is conceivable that a simple system, with all the SNMP parameters statically set and SNMP upcalls built into the UDP layer, may not require this.

```
int  snmp_agt_parse (u_char * data,
                     unsigned length ,
                     u_char * out_data,
                     unsigned out_length);
```

This is called by the transport glue layer whenever it has a datagram for the SNMP agent. Most commonly this means the UDP layer has received a datagram with a destination port of 161, SNMP's assigned number. The glue layer must strip the transport (UDP) headers from the received datagrams and pass them to `snmp_agt_parse()`, which is further described in the Function Descriptions section.

Port-dependent Files

Most of the work of porting SNMP is to modify or re-code the routines, definitions, and variables in the port files. These files are:

- snmp/snmpvars.c
- snmp/rfc1213.c
- snmp/rfc1213_.h
- snmp/snmpport.c
- h/snmpport.h
- snmpv1/v1sock.c
- snmpv1/v1port.c

MIB Compiler output files (based on MIBs supported) are `snmpvars.c` and `rfc1213_.h`. The file `rfc1213.c` contains implementation of MIB-II for InterNiche's TCP/IP stack. You would have to change /replace this file if you are using a different stack. File `snmpport.c` contains porting stuff common to all SNMP Agents (mainly `SNMPERROR()` and `GetUpTime()`). File `v1sock.c` is used when SNMPv1 over the Sockets, `SNMP_SOCKETS` option, is desired. Otherwise file `v1port.c` is used, `PREBIND_AGENT` option, to implement SNMPv1 over lightweight UDP API (of InterNiche).

Port-dependent changes in snmpport.h

The file `snmpport.h` contains definitions of a variety of SNMP limits, (longest datagram size, longest `Object Id` size, etc.) as well as definitions which bind the agent to the host system. This file is included in every `.c` source file in the agent sources. Definitions of such things as the protocols stack API and system library prototypes should be included here so as to insure that they are uniformly applied across the SNMP agent module.

The MIB Compiler output files are described in the MIB Compiler section of this document. Writing code to fill in the stubs is the majority of the work of implementing an SNMP agent.

The SNMP agent sources use a variety of definitions and C library calls. When possible, ANSI standards are used. However some of the portability functions are adapted from ad-hoc standards set by BSD. Since not all embedded system development environments support all these, the rest of this section lists and describes them in detail. Working examples of all these are included in the reference ports available from InterNiche.

`TRUE`, `FALSE`, and `NULL` must be defined. If not defined elsewhere, the examples below should work for almost every C environment:

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

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 and 32 bit quantities as show, and convert them from network format big-endian to the local CPU 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, as in this example:

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

`dtrap()` and `SNMP_ERROR()` are debugging aids. `dtrap()` is called by the SNMP 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 break point. 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 SNMP code will generally continue executing after a `dtrap()`, but the `dtrap()` usually indicates that something is wrong with the SNMP port.

NO PRODUCT BASED ON THIS CODE SHOULD BE SHIPPED UNTIL THE CAUSES OF ALL CALLS TO `dtrap()` HAVE BEEN INVESTIGATED AND ELIMINATED! When it comes time to ship code, the `dtrap()`s can be redefined to a null function to slightly reduce code size.

A data type for the individual sub-components of an SNMP `Object Id`s needs to be defined for each port. This type is named "`oid`", and is used throughout the SNMP code. Usually this is an unsigned 32 bit number, however applications which are extremely tight on space can shrink the `variables` table produced by the MIB compiler (and thus save considerable static data memory) by making this 16 or even 8 bits. (**Note:** Care should be taken if changing this size as it may cause problems with large enterprise IDs or TCP/UDP port numbers.) There should also be a definition of the maximum value which can be placed in a `Sub-Id`. The usual definitions are as follows:

```
typedef unsigned long oid; #define MAX_SUBID 0x7FFFFFFF
```

NOTE: Throughout the SNMP sources, the term `OID` is used to refer to both a `Sub-Id`, as above, or a complete SNMP `Object Id`.

Since the agent is written without any internal calls to `malloc()` or `free()` and needs to save small amounts of dynamic data somewhere, it has several compile time size limits. These are described below, along with recommended settings and examples.

The SNMP agent limits the maximum size, in bytes, required to hold an encoded SNMP `Object Id`. The macro for this is `MAX_OID_LEN`. The real limit required is determined by your MIB; the value of 64 assigned in this example is generous for most applications.

```
#define MAX_OID_LEN   64   /* max elements in an OID string */
```

There is a also a limit to the Maximum size a community string can grow. This example sets it at 32 bytes:

```
#define MAX_COMMUNITY_SIZE  32   /* max length in bytes of an community */
```

The example ports also put a size limit on the size of the various strings in the MIB-II System group. The ports could be re-written to dynamically allocate memory for arbitrarily large strings, but an engineer's life is much easier with a limit as follows:

```
#define SYS_STRING_MAX  256   /* max length of sys group strings */
```

Every company which ships an SNMP enabled product is supposed to have an SNMP Enterprise ID. This is a number assigned by the IANA (Internet Assigned Numbers Authority) which uniquely identifies each vendor of SNMP managed devices. You should obtain such a number before you ship your product. The contacts for reaching the IANA changes from time to time, but can be obtained from the latest "Assigned Numbers" RFC. If any case, to send traps or implement any basic MIB, the SNMP code will require an Enterprise ID. InterNiche customers are permitted to use the NetPort Software ID during product development, but they should obtain and recompile with their own prior to FCS. The definition is:

```
#define ENTERPRISE   18868   /* enterprise number */
```

Most ports limit the maximum size of a buffer an SNMP packet can occupy. While the best practical size will vary depending on your IP stack and media, 484 is always a safe minimum, and Ethernet or PPP based systems can usually use 1400.

```
#define SNMPSIZ   1400   /* MAX size of an snmp packet */
```

The last macro that porting engineers need to be aware of from `snmpport.h` defines the maximum number of TRAP targets the end user will be able to configure. This macro controls the size of a static table for TRAP target information.

```
#define MAX_TRAP_TARGETS 3
```

Compile-time Features and Options

There are various features which can be enabled via options in `ipport.h`. This section describes these options.

By far, the most important is the option `INCLUDE_SNMPV1`, which is used to enable the use of SNMPv1. This is done with the help of the following option. It enables all code in SNMPv1 directory.

```
#define INCLUDE_SNMPV1    1  /* SNMPv1 library, agent, & hook */
```

If SNMPv2c is to be used, then it can be enabled with the help of `INCLUDE_SNMPV2C` option. It enables all code in SNMPv2c directory and relevant code in SNMPv1 directory.

```
#define INCLUDE_SNMPV2C   1  /* SNMPv2c (community based SNMPv2) agent*/
```

`INCLUDE_SNMP` implements SNMP core (mib instrumentation, ASN parsing, etc). It should be enabled whenever any of SNMP agent (SNMPv1/SNMPv2c/SNMPv3) need to be used. To ensure this, the following macro is used. When InterNiche TCP/IP stack is used, this macro is used in `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 interfaces with UDP using sockets, the following option should be enabled.

```
#define SNMP_SOCKETS      1  /* SNMP over sockets, not lightweight API */
```

When InterNiche TCP/IP stack is used, this option is enabled in ipport.h. When the above option is enabled, code from v1sock.c is used. Otherwise code from v1port.c is used. InterNiche TCP/IP stack also supports lightweight API for UDP. Hence, in the ipport.h file, you will find the following.

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

With the help of the above, we say that if sockets is not used, then enable `PREBIND_AGENT`, so that SNMP can use the lightweight UDP API.

The InterNiche SNMP layer maintains packet counters as defined by MIB-II (RFC1213). For ports which don't want these counters, a small amount of space can be saved by omitting them. The counters are enabled with the following define:

```
#define MIB_COUNTERS 1
```

If you use SNMP traps, you need to include the trap code with `#defines` as follows:

```
#define ENABLE_SNMP_TRAPS 1
#define SNMP_UDP_TRAPS
```

TRAPs for non-IP ports can also been implemented on this code. In these cases you will want to use the first define but not the second.

Other Compile-Time Considerations

To reduce the codespace or dataspace requirements of your SNMP modules, the following suggestions are worth your consideration. Please note that there are situations where making these changes are not appropriate and you should be careful to thoroughly test your product after implementing any of these changes.

Filename	Description of Change
snmpport.h	Change the typedef of 'oid' from 'u_long' to 'u_short'. This will have a significant impact on the size of the variables[ ] table. Note that this change will restrict the maxumim OID sub-identifier supported by your target.
snmpport.h	Change the size of #define SYS_STRING_MAX to something smaller. This may impact the ability of an agent to set certain variables.
snmpport.h	Reduce the size of #define _MAX_PATH (see snmp_print_value() in snmputil.c)
snmpport.h	Redefine the macro SNMPERROR(msg) to a 'no-op'.
snmp_var.h	Change the size of #define SNMP_MAX_OIDS. This will result in savings of several Kbytes for each SNMPv2 and SNMPv3.
v3port.h	Setting V3_SHOW_ERR_MSG to 0 will reduce the size of SNMPv3 by several Kbytes, but will cause the errors to be presented by numeric values instead of strings.

3.2 The Target System

Most SNMP agents are ported to embedded systems, which are typically compiled on a DOS or UNIX workstation, burned into EPROM or flash, and installed in the target system for debugging. Sometimes an ICE or host resident development system is available to aid debugging and/or speed the operation of moving the system image to the target hardware. Exactly how these three routines will be implemented on your target system will depend on what type of OS and Network transport you have.

In commercial embedded systems like VRTX or PSOS, a separate task is usually created for the SNMP agent. Incoming SNMP packets are put in a message queue or mailbox by the network layer which is calling SNMP, and the SNMP task is scheduled to awaken. When awakened, the transport layer glue code dequeues the message and calls `snmp_agt_parse()`. When `snmp_agt_parse()` returns, the reply is passed back to the protocol stack as another message. If the InterNiche UDP/IP stack is used, it is generally implemented as a separate task.

Once the above routines have been implemented on the target system, some unit testing can be done on the SNMP agent. Some simple variables can be added to one of the `var_` routines, and a test packet sent to the SNMP stack. Most MIBs included the MIB-II system group. For testing purposes the code to implement this can be temporarily copied into the target system's `var_system()` stub. The SNMP stack should now answer `GET` and `GETNEXT` (and `GETBULK` for SNMPv2c) operations on `Object Ids` in the system group range.

If you have your protocol stack set up and an SNMP station is on your net, you can do a `GET` on `sysDescr`. You should probably have your ICE or debugger handy!

If you have not yet hooked up the protocol stack and wish to unit test the SNMP agent, there is a simple trick you can use. Hardcode the SNMP packet data into a static buffer, and pass a pointer to it to `snmp_agt_parse()`. With luck you will be rewarded with a well formed SNMP reply in your output data buffer. A code sample to help with this is included below. You should be aware that these values were typed in from a hexdump and may need modification for your system. The code below is provided for instruction only.

```
 unsigned char pkt1[ ] = {

/* ethernet header */

0x08, 0x00, 0x1a, 0x01, 0x02, 0x03,    /* dest address */
0x08, 0x00, 0x1a, 0x04, 0x05, 0x04,    /* src address */
0x08, 0x00,            /* IP ethernet-II type */

/* IP header */
0x45, 0x00, 0x00, 0x3f, 0x00,0x04, 0x00, 0x00, 0x1e, 0x11, 0x14, 0x9e,
0xc0, 0x09, 0xc8, 0x03,          /* source IP address */
0xc0, 0x09, 0xc8, 0x04,          /* dest IP address */

/* UDP header */
0x05, 0x72, 0x00, 0xa1, 0x00, 0x2b, 0x49, 0x24,

/* the SNMP portion for a GETNEXT on 1.3.6.1, community = "public" */
0x30, 0x21, 0x02, 0x01, 0x00, 0x04, 0x06, 0x70, 0x75, 0x62, 0x6c, 0x69,
0x63, 0xa1, 0x14, 0x02, 0x01, 0x2d, 0x02, 0x01, 0x00, 0x02, 0x01, 0x00,
0x30, 0x09, 0x30, 0x07, 0x06, 0x03, 0x2b, 0x06, 0x01, 0x05, 0x00
};

#define SNMPSIZ   484   // In real life this should usually be bigger
char snmpbuf[SNMPSIZ];   // buffer for SNMP code to build reply

void
testAgentParsing()
{
int reply_length;

   dtrap();   // hook debugger before we go in for trace

   reply_length = snmp_agt_parse(&pkt1[42], 35, &snmpbuf[0], SNMPSIZ);

   /* we should have gotten back a GET REPLY to sysDescr */
   dtrap();   // check reply here
}
```

3.3 GETNEXT/GETBULK and Indexes

This section is valid for both GETNEXT and GETBULK. Though GETNEXT is mentioned throughout this section, implementation for GETNEXT would work for GETBULK (used by SNMPv2c, SNMPv3) too.

Once the ported agent can generate a reply to a request, the remainder of the port, and most of the ongoing development, is in implementing the stub routines. The stub routines were discussed as MIB Compiler Output in the previous section. Before reading this section, you may want to go back and review that one, since this section and the next could be viewed as "advanced stub coding".

To review some SNMP basics, an "indexed" MIB variable is a variable which is part of a table, as opposed to the simpler "scalar" (non-indexed) variables. Scalar variables, such as the ones in the last chapter's examples, only occur once in an SNMP agent. For example, each agent only has one system group, and by extension, one sysDescr, one sysObject, one sysUptime, etc. On the other hand, Indexed variables (a.k.a. tables, or ASN.1 SEQUENCEs) can occur multiple times. For example, the MIB-II Interfaces table will have a complete set of interface variables for each network interface in the machine. A router with four Ethernet cards would have four ifIndexes, four ifDescrs, four ifPhysAddresses, etc. MIB-II stipulates that these are indexed by arbitrarily assigned numbers, in this case 1 through 4. The four ifDescr instances are represented as ifDescr.1, ifDescr.2, ifDescr.3, and ifDescr.4. The actual Object Ids are formed by appending the index (in this case as in most cases, an ASN.1 INTEGER) to the base Object Id.

An InterNiche stub routine for an indexed group has to do some extra work in addition to the work done by scalar variables: It has to determine if the Object Id passed has a valid index for the request. This gets a bit tricky when the request is a GETNEXT, for reasons which are outlined below. SETs are covered in the next section.

Below is the var_ifEntry() variables routine from the reference port, from file rfc1213.c, heavily annotated for this manual. It is the first routine in the file to deal with an indexed set of variables.

```
u_char *
var_ifEntry(
   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 */
{
```

Now, rather than simply checking for an exact match on GETs and SETs as in the earlier scalar examples, we must check to see if the index passed as the last component of the array pointed to by name (the Object Id from the actual received packet) is a valid index. ForGETorSETthis means it must exactly match one of our indexes. For GETNEXTs, it may be an exact match, or it may be an Object Id lexicographically lower than a valid response. We know it has at least a partial match of a valid interface table Object Id or the SNMP core code would not have called us with it. Since the SNMP core code does not know anything about our interface hardware, that information must be used here to put together the SNMP reply.

First, declare some scratch local variables. These include an `Object Id` buffer, `newname[ ]`, in which we will build trial `Object Id` names based on our interfaces. These will be passed to the `compare()` routine (a sort of `strcmp()` for `Object Ids`) to determine if we have a good candidate for a reply.

```
unsigned   interface;
oid        newname[MAX_NAME_LEN];
IFMIB      ifp;
int        result;
```

Start by copying the `Object Id` in the variables (`vp->name`) table into `newname[ ]`. `vp->name` contains the 'matching' entry in the `variables[ ]` table.

```
memcpy((char *)newname, (char *)vp->name, (int)vp->namelen * sizeof(oid));
```

For each interface in the machine, build an `Object Id` in `newname[ ]` with that interface's index and run it through `compare`.

```
/* find "next" interface */
   for(interface = 0; interface < ifNumber; interface++)
   {
      newname[10] = (oid)(interface + 1);
      result = compare(name, *length, newname, (int)vp->namelen);

      /* If the operation is a SET or GET and the object Ids match,
         break out of the loop */
      if(oper && (result == 0))
         break;
```

If the operation is a `GETNEXT` and newname is lexicographically greater than the received name, we have found the reply to the `GETNEXT`. Remember, when we got here the `vp->name` contains the 'matching' entry in the `variables[ ]` table.

By finding the first valid `Object Id` for an interface which is greater than the `Object Id` in the received packet, we have our `GETNEXT` reply.

```
if(!oper && (result < 0))
   break;
```

If we looped through all the interfaces without a match and return `NULL`, the SNMP core will try the `var_routine` for the next group or table.

```
if (interface >= ifNumber) return(NULL);
```

The rest of the routine is pretty much like examples for the scalar groups:

```
 memcpy((char *)name, (char *)newname, (int)vp->namelen * sizeof(oid));
*length = vp->namelen;
*var_len = sizeof(long);      /* default to 32 bit return */

ifp = nets[interface]->n_mib;

switch (vp->magic)
{
case IFINDEX:
   /* we store 'em 0 thru n-1, snmp wants 1 thru n */
   long_return = ifp->ifIndex + 1;
   return (u_char *)&long_return;
case IFDESCR:
   *var_len = strlen(ifp->ifDescr);
   return (u_char *)ifp->ifDescr;
case IFPHYSADDRESS:
   *var_len = nets[interface]->n_hal;
   return (u_char *)ifp->ifPhysAddress;
case IFLASTCHANGE:
   long_return = ((cticks * 100)/TPS);
   return (u_char*)&long_return;
case IFSPECIFIC:   /* could be Oid of ethernet MIB later. */
   *var_len = sizeof(oid00);
   return((u_char*)oid00);
default:   /* return 32 bit counter from table */
    return (u_char *)(((char *)ifp) + vp->magic);
}
```

Note that not all indexes are INTEGERS which are numbered sequentially as in the above example. A table could be indexed by any sequence of numbers. The Interfaces above could have been numbered 4, 5, 8, and 99. Then our for() loop would have had to look these up some how, perhaps even having to perform a sort first, or examining all the entries in the table and keeping a record of the best candidate.

3.4 Custom SET Operations

SET operations at their simplest are like GETs in that they usually require an exact match of the Object Id. They are more complex in two ways, however. The obvious complexity is that they need to set data or perform actions, rather than just report back to the SNMP core. The second wrinkle is that they may need to determine if a particular SET operation is legal. Possible problems include trying to SET an illegal value or a non-existent variable at the wrong time. The SNMP protocol defines a BAD_VALUE error for this occurrence.

The handling of the BAD_VALUE problem is the job of the var_ routine. If the var_ routine returns non-NULL, then it is assumed that it has filled in name, namelen, and the variable will be set. The actual setting occurs after the var_ routine returns to the SNMP core. This setting is done via one of two methods.

In a case where all that is required is a simple write to memory, the SNMP engine will write the value to be set (derived from the received SET packet) to the memory address returned by the var_ routine. Thus setting a INTEGER or OCTET STRING variable which is kept in a static location in memory involves no more work on the part of the programmer than a GET operation. (For strings, it is actually a good idea to check for buffer overflow. To do this easily, see the do_range flags described below.)

Sometimes it will be necessary to do more than just set an existing variable. You may need to send a message to another task, manipulate some hardware, or create a new entry in a table such as a route table. To accomplish this, InterNiche SNMP provides a single static structure called set_parms, defined in snmp_var.h:

```
struct SetParms {
   int (*access_method)(u_char *, u_char,int,u_char *, int);
   struct variable * vp;   /* pointer to variables table entry */
   oid  *name;       /* full Object ID of variable */
   int   name_len;   /* object count in "name". */
   int   do_range;   /* TRUE for range/length checking */
   long  hi_range;   /* high range for numeric sets */
   long  lo_range;   /* low  range for numeric sets */
   int   vbflags;    /* flags for special conditions like SF_NOASCIICHK
                      * cleared before processing every varbind */
   int   pduflags;   /* flags like SF_PDU_NEWROW, which are valid for PDU
                      * cleared before processing every pdu */
   struct SnmpObjects *vbinds; /* list of received varbinds in SET pdu */

#ifdef MIB_VALIDATION
   SIZE_INFO p_ranges; /* Set of ranges specified in MIB */
#endif
};
```

This structure consists of a pointer to a routine and a series of parameters for the routine.

The set_parms structure is cleared by the SNMP core prior to each SET call to a stub routine. If the var_ routine returns non-NULL, then the SNMP core checks to see if the routine pointer, access_method, has been set. If so, the routine is called INSTEAD of the regular simple setting of the memory location. This access_method routine has access to the fields in set_parms and to any other static variables that the var_ routine may have set.

For a working example, please see the atEntry() in the reference port's rfc1213.c file.

3.5 FAQ for implementing MIB objects

This section contains a list of frequently asked questions about MIB instrumentation. Please refer to this section for useful tips and guidelines to implementing MIB objects.

What set_parms is used for

The global set_parms is used to exchange information between MIB implementation and SNMP Engine. The MIB implementation in var_* functions can set the fields of set_parms to convey special information to the SNMP Engine. All fields of set_parms are reset before processing every varbind. Hence, the settings done for one object don't affect another object. set_parms.pduflags is reset before processing a new PDU. Hence any change to set_parms.pduflags will be valid till the PDU is being processed. As of this writing, set_parms.pduflags is only used during new row creation. Here is some description about individual fields of set_parms.

- When a special function needs to be used for doing a SET, `set_parms.access_method` is initialized.
- When the length of a string object needs to be validated during SET, `set_parms.do_range`, `set_parms.hi_range` and `set_parms.lo_range` are initialized.
- When the range of an integer object needs to be validated during SET, `set_parms.do_range`, `set_parms.hi_range` and `set_parms.lo_range` are initialized.
- When specific checks are to be done for received values, then `set_parms.vbflags` is initialized.
- When a new row is being created, set_parms.pduflags is initialized.
- The `set_parms.vbinds` field is initialized by the SNMP Engine and must not be changed by the `var_*` function. It can be used by the var_* function to determine information about other varbinds in the current PDU.
- The use of `set_parms.p_ranges` is internal to SNMP engine and this field must not be used in any `var_*` function.

Magic numbers and their scope

Any `var_*` function can implement a number of objects. The magic number is used to determine the object under scrutiny. The use of magic number is limited to the MIB instrumentation. That is, the SNMP Engine doesn't know/care about it. If you search for "magic", you will find that the only files using it are `rfc1213.c` and `v3mib.c`.

How to implement support for a simple Integer object

Please refer to Chapters 2, 3 for a detailed description. To implement a simple Integer object, you will need to the following in the `var_*` function

Update name of OID	For a scaler object, this would be as simple as `MEMCPY(name, vp->name, (int)vp->namelen * sizeof(oid));`
Update length of OID	For a scaler object, this would be as simple as: `*length = vp->namelen;`
Update var_len (length of value)	`*var_len = sizeof(long);  /* default length */`
Return a pointer to the actual object	`return (u_char *)&long_value;`

For example, to implement `SysServices` object in `var_system()`, we have the following code

```
MEMCPY(name, vp->name, (int)vp->namelen * sizeof(oid));
*length = vp->namelen;
*var_len = sizeof(long);
switch (vp->magic)
{
case SYSSERVICES:
   return (u_char *)&sys_services;
}
```

How to implement support for a simple string object

Please refer to Chapters 2, 3 for a detailed description. To implement a simple string (`OCTETSTRING /DisplayString`) object, you will need to the following in the `var_*` function

Update name of OID	For a scaler object, this would be as simple as `MEMCPY(name, vp->name, (int)vp->namelen * sizeof(oid));`
Update length of OID	For a scaler object, this would be as simple as `*length = vp->namelen;`
Update var_len (length of value)	`*var_len = strlen(string_value);`
Return a pointer to the actual object	`return (u_char *)string_value;`

For example, to implement `SysDescr` object in `var_system()`, we have the following code

```
MEMCPY(name, vp->name, (int)vp->namelen * sizeof(oid));
*length = vp->namelen;
switch (vp->magic)
{
   case SYSDESCR:
   *var_len = strlen(sys_descr);
   return (u_char *)sys_descr;
}
```

How to validate length of string during SET

If `MIB_VALIDATION` is used and the object's length is specified in MIB, then the SNMP engine would validate the length of incoming string value. However, if a check is desired then set the range in the `set_parms` global. For example, if a `string` is defined by char `str[256]`, then the following lines will enforce that the received value is in range.

```
set_parms.do_range = TRUE;
set_parms.hi_range = 255;
set_parms.lo_range = 0;
```

As strings are terminated by 0, the hi_range value should be 1 less than the max size.

How the received values are validated

If `MIB_VALIDATION` is enabled, then variables[ ] would contain the validation information, as mentioned in the MIB. And at runtime `validateValue()` is called to validate the received value.

- If `MIB_VALIDATION` is disabled then the above validation will not be done.
- Even if `MIB_VALIDATION` is enabled, if no information is present in MIB, then no validation will be done.
- Using `set_parms.do_range`, specific validation can be done for an object.
- If `set_parms.do_range` is enabled for an object, then validation is done accordingly. If the received value passes this test, then the default validation (as defined by MIB_VALIDATION) will be done.
- If `set_parms.access_method` is defined, then the object will be using a custom "SET function" to set the value. In this case, default validation (as defined by MIB_VALIDATION) is not done.

How to check if the received string complies with US ASCII character set

SNMP Engine will do this for you. That is, for all string objects, `is_us_ascii()` is called to check compliance with US ASCII character set. If a string doesn't pass this test, then the appropriate error is reported to the SNMP Manager.

What to if you do not want to do US ASCII compliance test for an object

If US ASCII compliance test is to be skipped for an object, then `SF_NOASCIICHK` flag should be set for `set_parms.vbflags` (in the `var_*` function). Certain objects (for example ethernet addresses) are received as `OCTETSTRINGS`, but their values are not bound by US ASCII character set. Hence, to skip the test for these objects, the `SF_NOASCIICHK` flag should be set. For example, here is how it is done for `atNetAddress` object.

```
set_parms.vbflags |= SF_NOASCIICHK ; /* don't do is_us_ascii() check */
```

How to ask the SNMP Engine to validate an unsigned object

If an object can only have unsigned values, the SF_UNSIGNEDVALUE flag can used to ask the agent to validate it. For example, SnmpTargetAddrTimeout object can only have unsigned values. Hence the following initialization is done in the var_* function.

```
set_parms.vbflags |= SF_UNSIGNEDVALUE ;
```

When SF_UNSIGNEDVALUE flag is enabled, SNMP engine would check whether the received value is signed or not. If it is signed/negative, then the appropriate error is reported to the SNMP Manager.

How to ask the SNMP Engine to validate a StorageType object

For a StorageType object, the SF_STORAGETYPE flag can used to ask the agent to validate it. For example, SnmpTargetStorageType has the following initialization (in the var_* function).

```
set_parms.vbflags |= SF_STORAGETYPE;
```

When SF_STORAGETYPE flag is enabled, SNMP engine would check whether the received value is valid or not. If it is not valid, then the appropriate error is reported to the SNMP Manager.

How to ask the SNMP Engine to validate a SecurityLevel object

For a SecurityLevel object, the SF_SECURITYLEVEL flag can used to ask the agent to validate it. For example, SnmpTargetParamsSecurityLevel has the following initialization (in the var_* function).

```
set_parms.vbflags |= SF_SECURITYLEVEL;
```

When SF_SECURITYLEVEL flag is enabled, SNMP engine would check whether the received value is valid or not. If it is not valid, then the appropriate error is reported to the SNMP Manager.

How to report a NoCreation error to SNMP Manager

A new row is created in the var_* function. If there is an error in creating the new row and NoCreation needs to be reported to the SNMP Manager, then enable the SF_NOCREATION flag set_parms.vbflags.

For example, in vacmAccessTable, when the received indices are wrong, we send NoCreation error to SNMP Manager by setting the SF_NOCREATION flag.

```
set_parms.vbflags |= SF_NOCREATION;
```

How to report WrongValue error from var_* function

More specifically: For a request for a new row, if the value of RowStatus object is wrong, how do I report wrongValue error to SNMP Manager?

This is a very special and rare requirement. A new row is created in the var_* function. If the RowStatus object's value is wrong (it should be CreateAndGo or CreateAndWait or Destroy) and wrongValue needs to be reported to the SNMP Manager, then the SF_WRONGVALUE flag set_parms.vbflags should be enabled. For example, in var_vacmSecurityToGroupEntry, when we receive NotReady value (while creating new row), we set the following.

```
set_parms.vbflags |= SF_WRONGVALUE;
```

The error codes that can be returned by custom SET functions

The custom SET functions are those which are used via set_parms.access_method.

Any of the V3_VB_ error codes defined in npsnmp.h can be returned. For example V3_VB_WRONGVALUE, V3_VB_INCONSISTENTVALUE, etc. If SNMPv1 agent is doing the processing, then it would only report SNMPv1 errors. That is, it would map the SNMPv2c/SNMPv3 errors to SNMPv1 errors (as defined in the coexistence RFC).

In the var_* function, how to get preview of all varbinds in the SET PDU

The set_parms.vbinds data structure can be used in var_* function to get a preview of all varbinds present in current SET PDU. Preview is generated (and needed) only for a SET request.

How to find out the number of varbinds in the var_* function

When the var_* function is called during a SET request, the set_parms.vbinds data structure is initialized. The set_parms.vbinds->num field contains the number of varbinds present in PDU.

What to do to implement creation of a new row

To support creation of a new row, following things need to be done.

- Find out the when to create a new row. That is, in the var_* function, when we can't find a row matching the incoming OID, and if it is a SET operation, we need to create a new row
- Set the SF_PDU_NEWROW flag in set_parms.pduflags
- Take a peek at the received RowStatus value to do a sanity check. Only create row if the received values are CreateAndGo or CreateAndWait.
- For CreateAndGo, find out if all the mandatory objects are present.

- For `CreateAndWait`, find out when all the mandatory objects have been initialized. At that point, transition the row from `notReady` to `notInService`.
- Whenever a `SET` is done on the `RowStatus` object, use `snmp_update_rowstatus()` to update the value.

How to implement CreateAndGo row creation

When in a `SET` request the `RowStatus` object has the value `CreateAndGo`, a new row should be created and made active. If some mandatory fields are required for the creation of new row, then the new row should be created only if all of those are present. Also, a new row should be created only if all these objects have valid values.

To accomplish this, when the row is created in `var_*` function, the `SF_PDU_NEWROW` flag in `set_parms.pduflags`. Then, when error occurs in any value of mandatory row objects, the `SF_PDU_NEWROW` flag can be checked. If this flag is set, then the current entry is deleted. Using this mechansim, we ensure that if any error occurs, we don't leave an extraneous entry in the table.

For some tables, just a `SET` with the `RowStatus` object is sufficient to create a new row. For other tables, some other objects may be required to create a new row. in such cases, we can do a sanity check before creating a new row in `var_*` function.

That is, by looking at `set_parms.vbinds->num`, we can find out the number of varbinds in current PDU. If the number of varbinds is less than number of mandatory objects, we can conclude that there is insufficient info and hence avoid the row creation.

To implement the requirement of mandatory objects, a flags field should be used in the table. For example, the `flags` field is used in `UserTable` for ensure that all mandatory objects are present during a request for a new row creation.

How to implement CreateAndWait row creation

When in a`SET`request the `RowStatus` object has the value `CreateAndWait`, a new row should be created and made `notReady`. When all the mandatory values have been set, then the status should be made `notInService`.

To implement the requirement of mandatory objects, a `flags` field should be used in the table. For example, the `flags` field is used in `UserTable` for ensure that all mandatory objects are present during a request for a new row creation.

When all the values have been received, the status is changed from `notReady` to `notInService`.

How to do a sanity check to avoid unnecessary creation of a row

More specifically: The manager tries to create a new row with `RowStatus` object of value "`DESTROY`". How to do a sanity check to avoid unnecessary creation of row?

Here is what happens normally.

1. Request for a new row is received.
2. `var_*` function is called, which creates a new row and returns a pointer. Usually it sets `set_parms.access_method`.
3. SNMP engine calls `set_parms.access_method` to set the value.

When the received value is 6 (that is "`DESTROY`"), then a row gets unnecessarily created in step 2 above. Then the row gets deleted in step 3.

By doing a sanity check in `var_*` function, it is possible to avoid this unnecessary row creation and deletion. Here is some sample code which does this check (used in `usmUserTable`).

```
if (set_parms.vbinds)
{
   /* if request has only 1 oid and its RowStatus, process value as per RFC */
   if ((set_parms.vbinds->num ==1) && (vp->magic == USMUSERSTATUS))
   {
      /* take a peek at the rcvd value */
      if (*(set_parms.vbinds->objs[0].value) == SNMP_RS_DESTROY)
      {
         /* no action needed at agent - return noError to imply
          * that a new row was created and destroyed */
         *var_len = sizeof(long);   /* default length */
          return (u_char *)&long_return;
      }
   }
}
```

4 Function Descriptions

4.1 SNMP Agent Interface

These two routines are only SNMP agent routines that most programmers porting InterNiche SNMP will ever need to call directly. They are the whole external interface to the portable SNMP agent.

snmp_agt_parse

Name

`snmp_agt_parse`

Syntax

```
int snmp_agt_parse(u_char * inbuf, unsigned inlength, u_char * outbuf,
unsigned outlength);
```

Parameters

`nbuf`	pointer to beginning of SNMP datagram
`inlength`	length of inbuf data
`outbuf`	pointer to buffer for possible reply
`outlength`	length of outbuf

Description

`snmp_agt_parse()` is called by the transport stack (or an implementation routine on top of the transport stack) when an SNMP datagram for the SNMP agent is received. An example of this would be a UDP packet with destination port 161. **This routine is the sole entry point to the agent from the protocol stack**. The incoming SNMP data is passed in the pointer `inbuf`, with the length of the datagram in `inlength`. `inbuf` should point to the first byte of the ASN.1 data (usually `0x30`) and not to a UDP (or other transport) header.

All processing of the SNMP datagram is done during the call to `snmp_agt_parse()`. `snmp_agt_parse()` may leave a reply to the datagram in the buffer indicated by `outbuf`, so it should be big enough to hold any expected reply. According to the SNMP RFCs the minimum size of this buffer should be 484 bytes. Larger values (e.g., SNMPSIZ (1400)) are recommended when they can be supported by the system architecture and network topology.

If `snmp_agt_parse()` returns a non-zero length, then the calling code should make sure the reply data in outbuf is sent back to the party which sent the SNMP request. On UDP this means preserving the incoming port value and IP address.

Returns

Returns 0 if there is no reply data in `outbuf`, else returns the number of bytes in the SNMP reply in `outbuf`.

snmp_trap

Name

`snmp_trap`

Syntax

```
int
snmp_trap(
   int   trapType,                /* other parms are ignored if trapType != 6 *

   int   specificType,            /* vendor TRAP if trapType == 6 */
   int   specificVarCount,        /* number of variables */
   struct trapVar * specificVars, /* array of variable info */
   int   version)                 /* SNMP version (v1/v2c) */
```

Parameters

`trapType`	One of the predefined SNMP traps, in the range of 1-6. If `trapType` is not 6 (vendor specific trap), the remaining variables are ignored by SNMP and may be `0` or `NULL`.
`specificType`	a vendor specific type. These are defined by the vendor.
`specificVarCount`	number of entries in `specificVars`.
`specificVars`	pointer to a array of `trapVar` structures.

Description

snmp_trap() takes the passed parameters, and builds and sends a SNMP trap. Based on the version number, it will send a SNMP v1 TRAP or a SNMP v2 TRAP. The trap is built in a static buffer provided at snmp_init() time. A TRAP packet is sent to each of the hosts in the trap_targets[ ] table.

The structure trapVar is defined in file snmp_imp.h. The current version is reproduced here, but please refer to the file version when writing code.

```
struct trapVar { /* struct for each trap variable */
  oid        varName[MAX_OID_LEN];   /* Object Id of variable */
  unsigned   varNameLen;             /* oid components in varName */
  u_char     varType;                /* ASN.1 type of variable */
  unsigned   varValLen;              /* octets in variable data field */
  u_char *   varBuf;                 /* the actual variable data */
  unsigned   varBufLen;              /* used only by snmp_parse_trap() */ };
```

To send a vendor specific a TRAP with variables attached, you will need to allocate (either statically or dynamically) space for an array of these structures, one per variable. You will then need to fill in the values for your TRAP variables prior to calling snmp_trap(). Once the snmp_trap() call returns, the trapVar array may be freed or reused. InterNiche provides a sample example in the form of snmp_test_sp_trap(). For testing it, SNMP_TEST_SP_TRAP must be enabled in snmpport.h

Returns

Returns 0 if parse error, else returns length of trap image built in passed buffer.

4.2 User Required Functions

The functions in this section must by provided by the porting engineer as part of the port work. Most are referenced at other places in the manual and are described in detail here.

SNMPERROR

Name

`SNMPERROR`

Syntax

```
SNMPERROR( char * )
```

Parameters

`msg`	C String text of error message to print.

Description

The `SNMPERROR()` routine is called by the SNMP code when it detects an error specific to the SNMP protocol. This can be the result of a badly formed packet received from the net, so it is not practical to completely eliminate calls to this routine. This is meant to print messages for the programmer's benefit during product development. It can be #ifdefed out before shipping, or its output can be directed to some sort of error log or user console. On systems which support `printf()`, `SNMPERROR()` can be `#ifdefed` to `printf()`.

Returns

Returns no meaningful value.

Example`#define SNMPERROR(msg) snmp_error(msg)`

send_trap_udp

Name

`send_trap_udp`

Syntax

```
int send_trap_udp(u_char * out_data, int out_len, ip_addr trap_target);
```

Parameters

`out_data`	pointer to SNMP TRAP packet image to send.
`out_len`	length of data pointer to by out_data above.
`trap_target`	32 bit IP address of host to send trap to.

Description

This is called by the SNMP core code to send the standard traps, such as Authentication failure. Since `send_trap_udp()` is called from the SNMP core, the name and function parameters should not be changed. This routine simply has to send the trap buffer passed to the IP address specified at UDP port 162. It is usually a very small routine. Non UDP systems can implement this by ignoring the IP address passed and send the trap buffer on the appropriate interface.

Returns

Should return 0 if OK, else -1. The standard SNMP agent code ignores this return, however special ports have been implemented which take other action (e.g. paging an operator) when trap sends fail.

GetUptime

Name

`GetUptime`

Syntax

```
u_long GetUptime(void);
```

Parameters

none

Description

This routine is called from the SNMP agent trap generation code to timestamp outgoing TRAPs. Ports which implement MIB-II also use it for the `sysUptime` variable in the system group.

Returns

Returns a 32 bit unsigned value containing the number of SNMP TIMETICKS (100th seconds intervals) since the system was last rebooted.

Example

```
u_long
GetUptime()
{
   return ((CTICKS * 100L) / TPS );
}
```

5 Command Line Interface

For those systems supporting a command line interface a number of interactive commands are provided to help the porting engineer examine, modify and test the internal workings of the SNMP agent. The commands can be invaluable both during debugging of the server and to the end user during configuration and runtime. If you do not implement these menu commands as provided, we strongly suggest that some alternative method (i.e. a GUI) be provided to the end user for accessing the same data.

5.1 snmp community

Command Name

`snmp community - manage the SNMP community table`

Syntax

```
community -i INT -c STRING -a STRING

community -c STRING -a STRING -n STRING -e STRING [-k STRING -v STRING -t
INT -s INT]

community -d INT
```

Parameters

`-i`	Argument of type `INT`, specifying the SNMPv1/SNMPv2c community table index.
`-c`	Argument of type `STRING`, specifying the community string.
`-a`	Argument of type `STRING`, specifying the access permissions.
`-n`	Argument of type `STRING`, specifying the Security Name associated with the SNMPv3 table entry.
`-e`	Argument of type `STRING`, specifying the Engine ID associated with the SNMPv3 table entry.
`-k`	Argument of type `STRING`, specifying the Context Name associated with the SNMPv3 table entry.
`-v`	Argument of type `STRING`, specifying the Tag value associated with the SNMPv3 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 table entry to delete.

Description

This command manages the table entries for the SNMPv1/SNMPv2c or SNMPv3 community table. The `'-i'` parameter selects the SNMPv1/SNMPv2c community table. If `'-i'` is absent, the SNMPv3 community table is assumed. Table indexes begin at 1. Possible values for the access strings are: `RONLY, RWRITE, NOACCESS, READCREATE, WRITEONLY, and NOTIFY`.

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

Notes/Status

- NOTIFY is equivalent to ACCESSIBLE_FOR_NOTIFY.
- If the community string is too long, it is truncated.
- The `'-k'` parameter defaults to the default context name.
- The `'-t'` parameter defaults to 4 = permanent.
- The `'-s'` parameter defaults to 1 = active.

Location

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

5.2 snmp config

Command Name

`snmp config - configure SNMP agent`

Syntax

```
config [-a] [-c STRING] [-d STRING] [-k STRING] [-n STRING] [-l STRING] [-b INT]
```

Parameters

`-a`	(no parameter), toggle the SNMP Agent ON/OFF.
`-c`	Argument of type `STRING`, specifying the default community string.
`-d`	Argument of type `STRING`, stored in the `sysDescr` field of the `RFC1213-MIB system` definition.
`-k`	Argument of type `STRING`, stored in the `sysContact` field of the `RFC1213-MIB system` definition.
`-n`	Argument of type `STRING`, stored in the `sysName` field of the `RFC1213-MIB system` definition.
`-l`	Argument of type `STRING`, stored in the `sysLocation` field of the `RFC1213-MIB system` definition.
`-b`	Argument of type `INT`, specifying the SNMPv3 engine boot count.

Description

This command is used to set various fields in the RFC1213-MIB structure. If the command is successful, the updated values are displayed.

Notes/Status

- The current SNMP configuration is displayed after any fields have been updated.
- String values which are too long will be truncated.
- A warning is displayed if the new engine boot count is less than the current value.
- The sysDescr.0 MIB variable can't be updated via SNMP (because it's categorized as read-only).

Location

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

5.3 snmp mib

Command Name

`snmp mib - display SNMP MIB counters`

Syntax

`mib`

Description

This command displays SNMP MIB counters.

Location

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

5.4 snmp netstat

Command Name

`snmp netstat - display SNMP agent information`

Syntax

`netstat`

Description

Displays information related to the SNMP Agent.

Notes/Status

- SNMPv3 Agent information is displayed when `INCLUDE_SNMPV3` is defined.

Location

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

5.5 snmp target

Command Name

`snmp target - configure SNMP trap target`

Syntax

```
target -i INT [-a IPADDR] [-c STRING] [-d]
```

Parameters

`-i`	Argument of type `INT`, indicating `the trap target entry to modify`.
`-a`	Argument of type `IPADDR`, indicating `the IPv4 or IPv6 address of the trap target`.
`-c`	Argument of type `STRING`, indicating `the community string associated with the trap target`.
`-d`	(no parameter), delete the selected trap target.

Description

This command configures a trap target entry. An entry in the trap target table contains a destination IPv4 address and a community string. When a trap message is sent, it is sent to each entry in the trap table.

Notes/Status

- Trap target indexes range from 1 to MAX_TRAP_TARGETS.
- If the community string is too long, it will be truncated.
- The '-d' parameter takes precedence over all other parameters.
- If no parameters are specified, the current trap target table is displayed.

Location

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

5.6 snmp trap

Command Name

`snmp trap - send SNMP trap(s)`

Syntax

```
trap -v {1,2} (-a | -t INT)

trap -v 3 (-a | -k STRING) -c STRING -k STRING
```

Parameters

`-v`	Argument of type `INT`, indicating the SNMP version, range is 1..3.
`-t`	Argument of type `INT`, indicating the type of trap message to send, range is 0..6.
`-a`	(no parameter), send a trap message for each trap type.
`-k`	Argument of type `STRING`, indicating the tag value of the corresponding target table entries.
`-c`	Argument of type `STRING`, specifying the context name.

Description

Sends a `SNMP TRAP` message to each trap target. The trap message is formatted using the specified trap type and SNMP protocol version. If '-a' is specified, a trap message of each type is sent. An error is returned if the selected SNMP version is not supported.

Notes/Status

- The `'-v'` parameter defaults to SNMPv1.
- If both `'-t'` and `'-a'` are specified, `'-a'` is assumed.
- If the trap type is 6 and `SNMP_TEST_SP_TRAP` is defined, a second "enterprise-specific" trap of type 6 is sent.
- The community string defaults to the global community string (see the `snmp config` command).

Location

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