HEWLETT® PACKARD

SNMP++

C++ Based Application Programmers Interface for the Simple Network Management Protocol

This document describes SNMP++, an open specification for object oriented network management development using SNMP and C++. This document describes the various portable classes which make up the API and goes through a number of examples.

Visit the SNMP++Web site for source code, real working examples and other related documents @ http://rosegarden.external.hp.com/snmp++

Peter Erik Mellquist
Hewlett-Packard Company
Workgroup Networks Division
Network Management Section
Roseville, CA

Hewlett-Packard Company version 2.61 11 July, 1997

Hewlett Packard
Peter Erik Mellquist

Author's Contact Information:
Comments, suggestions and inquiries regarding SNMP++ may be submitted via electronic mail to peter_mellquist@hp.com.

Acknowledgments:
I would especially like thank five people for their generous assistance. For revision 2.6 and all of its changes, credit is given to Brian O'Keefe of HP Network Management Systems Division, for his many suggestions and SNMP version 2 knowledge. The SNMP++ project would not have been possible if not for Kim Banker's , HP Roseville Networks Division, continual support and effort. Bob Natale of American Computers and Electronics Corp., provided valuable access the WinSNMP working group where many comments and suggestions have originated. Lastly Jeff Meyer and Tom Murray, of HP Network Computing Division, provided significant contributions to the classes and the UNIX implementation of the code base.

Where this document and code can be found:
This document is freely available in both Microsoft Word for Windows and Postscript formats on the following WWW server.

http://rosegarden.external.hp.com/snmp++

Technical Contributors:
Kim Banker , HP Roseville Networks Division
Gary Berard , HP Roseville Networks Division
Chuck Black ,HP Roseville Networks Division
Bruce Falzarano ,HP Roseville Networks Division
Greg Fichtenholtz , HP OpenView Operations
Harry Kellog, HP Roseville Networks Division
Moises Medina , HP Roseville Networks Division
Jeff Meyer, HP Network Computing Division
Tom Milner, HP Roseville Networks Division
Tom Murray, HP Network Computing Division
Mark Pearson, HP Roseville Networks Division
Bob Natale, American Computers and Electronics Corporation
Brian O'Keefe, HP Network Systems Management Division
Frank Wang, Purdue University

Table Of Contents

1. What's New in Revision 2.61

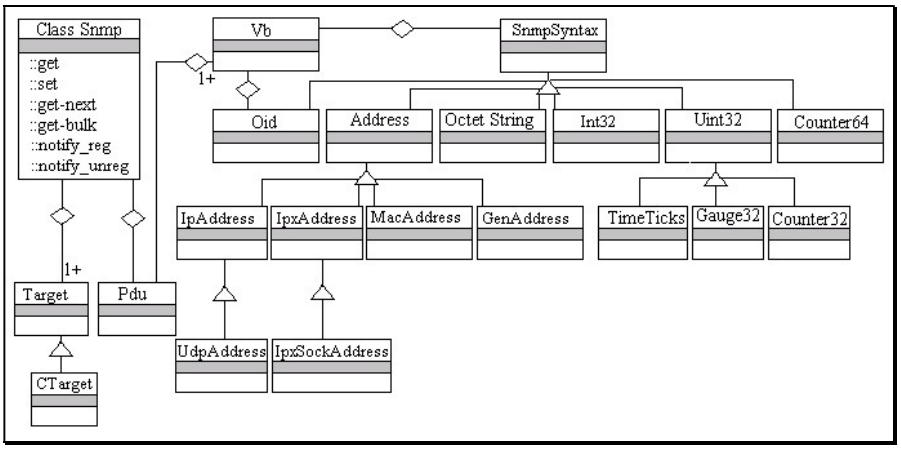
Version 2.6 is 100% compatible with version 2.5. Version 2.6 includes a number of minor enhancements plus a few bug fixes.

- UNIX classes include source code for full ASN.1 encoding and decoding
- ASN.1 encoding and decoding wrapped into SnmpMessage class
- Minor bug fixes
- Dropped support for Win16
- Full Support for WinSNMP 2.0

2. Products Now Using SNMP++

- *HP Download Manager IP & IPX for MS-Windows 3.1, 3.11, NT and Win '95*
- *HP Download Manager for HPUX 9.0 and 10.0*
- *HP Download Manager for Sun Solaris*
- *HP Router Monitor For OpenView MS-Windows 3.1, 3.11, NT and Win '95*
- *HP Router Monitor for OpenView HPUX 9.0*
- *HP InterConnect Manager for OpenView MS-Windows 3.1, 3.11, NT and Win '95*
- *HP InterConnect Manager For OpenView HPUX 9.0 and 10.0*
- *HP Virtual LAN Switch Configurator For OpenView MS-Windows*
- *HP Virtual LAN Switch Configurator For OpenView HPUX*
- *SNMP++ Demonstration Application For MS-Windows, Win16 and Win32*
- *Fiber Channel Switch Manager for MS-Windows 3.1, 3.11, NT and Win '95*
- *Fiber Channel Switch Manager for HPUX 9.0 and 10.0*
- *HP Advance Stack Assistant For Windows*
- *HP Advance Stack Assistant For HPUX*
- *HP OpenView Professional Suite For Windows*

Object Modeling Technique (OMT) view of the SNMP++ Framework



3. Introduction

Various Simple Network Management Protocol (SNMP) Application Programmers Interfaces (APIs) exist which allow for the creation of network management applications. The majority of these APIs provide a large library of functions which require the programmer to be familiar with the inner workings of SNMP and SNMP resource management. Most of these APIs are platform specific, resulting in SNMP code specific to an operating system or network operating system platform and thus not portable. Application development using C++ has entered the main stream and with it a rich set of reusable class libraries are now readily available. What is missing is a standard set of C++ classes for network management. An object oriented approach to SNMP network programming provides many benefits including ease of use, safety, portability and extensibility. SNMP++ offers power and flexibility which would otherwise be difficult to implement and manage.

3.1. What Is SNMP++?

SNMP++ is a set of C++ classes which provide SNMP services to a network management application developer. SNMP++ is not an additional layer or wrapper over existing SNMP engines. SNMP++ utilizes existing SNMP libraries in a few minimized areas and in doing so is efficient and portable. SNMP++ is not meant to replace other existing SNMP APIs such as WinSNMP, rather it offers power and flexibility which would otherwise be difficult to manage and implement. SNMP++ brings the ***Object Advantage*** to network management programming.

3.2. Objectives of SNMP++

3.2.1. Ease of Use and SNMP++

An Object Oriented (OO) approach to SNMP programming should be easy to use. After all, this is supposed to be a simple network management protocol. SNMP++ puts the simple back into SNMP! The application programmer does not need be concerned with low level SNMP mechanisms. An OO approach to SNMP encapsulates and hides the internal mechanisms of SNMP. In regard to ease of use, SNMP++ addresses the following areas.

3.2.1.1. Provides an easy-to-use interface into SNMP

A user does not have to be an expert in SNMP to use SNMP++. Furthermore, a user does not have to be an expert in C++! For the most part C pointers do not exist in SNMP++. The result is an easy to use straight forward API.

3.2.1.2. Provides easy migration to SNMP version 2

A major goal of SNMP++ has been to develop an API which scales to SNMP version 2 with minimal impact on code. The SnmpTarget class makes this possible.

3.2.1.3. Preserves the flexibility of lower level SNMP programming

A user may want to bypass the OO approach and code directly to low level SNMP calls. SNMP++ is fast and efficient. However, there may be instances where the programmer requires coding directly to a lower level SNMP API.

3.2.1.4. Encourages programmers to use the full power of C++ without chastising them for not learning fast enough

A user does not have to be an expert in C++ to use SNMP++. Basic knowledge of SNMP is required, but as will be shown, a minimal understanding of C++ is needed.

3.2.2. Programming Safety and SNMP++

Most SNMP APIs require the programmer to manage a variety of resources. Improper allocation or de-allocation of these resources can result in corrupted or lost memory. SNMP++ provides safety by managing these resources automatically. The user of SNMP++ realizes the benefits of automatic resource and session management. In regard to programming safety, SNMP++ addresses the following areas.

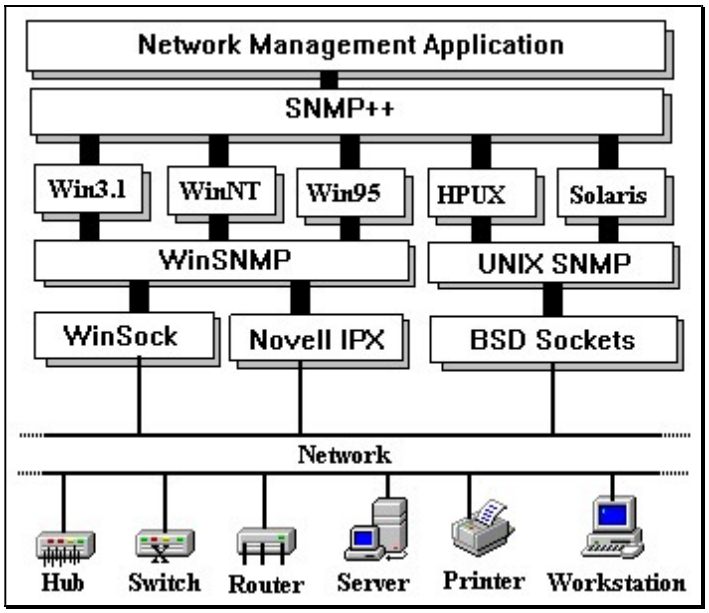
3.2.2.1. Provides safe management of SNMP resources

This includes SNMP structures, sessions, and transport layer management. SNMP classes are designed as Abstract Data Types (ADTs) providing data hiding and the provision of public member functions to inspect or modify hidden instance variables.

3.2.2.2. Provides built in error checking, automatic time-out and retry

A user of SNMP++ does not have to be concerned with providing reliability for an unreliable transport mechanism. A variety of communications errors can occur including: lost datagrams, duplicated datagrams, and reordered datagrams. SNMP++ addresses each of these possible error conditions and provides the user with transparent reliability.

3.2.3. Portability and SNMP++



A major goal of SNMP++ is to provide a portable API across a variety of operating systems (OSs), network operating systems (NOSs), and network management platforms. Since the internal mechanisms of SNMP++ are hidden, the public interface remains the same across any platform. ***A programmer who codes to SNMP++ does not have to make changes to move it to another platform***. Another issue in the area of portability is the ability to run across a variety of protocols. SNMP++ currently operates over the Internet Protocol (IP) or Internet Packet Exchange (IPX) protocols, or both using a dual stack.

3.2.4. Extensibility and SNMP++

Extensibility is not a binary function but rather one of degree. SNMP++ not only can be extended, but can and has been extended easily. Extensions to SNMP++ include supporting new OS's, NOS's , network management platforms, protocols, supporting SNMP version 2, and adding new features. Through C++ class derivation, users of SNMP++ can inherit what they like and overload what they wish to redefine.

3.2.4.1. Overloading SNMP++ Base Classes

The application programmer may subclass the base SNMP++ classes to provide specialized behavior and attributes. This theme is central to object orientation. The base classes of SNMP++ are meant to be generic and do not contain any vendor specific data structures or behavior. New attributes can be easily added through C++ sub-classing and virtual member function redefinition.

4. An Introductory Example

Rather than begin by describing SNMP++ and all of its features, here is a simple example that illustrates its power and simplicity. This example obtains a SNMP MIB System Descriptor object from the specified agent. Included are all code needed to create a SNMP++ session, get the system descriptor, and print it out. Retries and time-outs are managed automatically. The SNMP++ code is in bold font.

4.1. A Simple SNMP++ Example

```
#include "snmp_pp.h"
#define SYSDESCR "1.3.6.1.2.1.1.1.0"              // Object ID for System Descriptor
void get_system_descriptor()
{
  int status;                                     // return status
  CTarget ctarget( (IpAddress) "10.4.8.5");       // SNMP++ community target
  Vb vb( SYSDESCR);                               // SNMP++ Variable Binding Object
  Pdu pdu;                                        // SNMP++ PDU

  //-------[ Construct a SNMP++ SNMP Object ]-------------------------------------
  Snmp snmp( status);                             // Create a SNMP++ session
  if ( status != SNMP_CLASS_SUCCESS) {            // check creation status
    cout << snmp.error_msg( status);              //  if fail, print error string
    return;  }

  //-------[ Invoke a SNMP++ Get ]------------------------------------------------
  pdu += vb;                                      //  add the variable binding to the PDU
  if (  (status = snmp.get(  pdu, ctarget)) != SNMP_CLASS_SUCCESS)
    cout << snmp.error_msg( status);
  else {
    pdu.get_vb( vb,0);                            // extract the variable binding from PDU
    cout << "System Descriptor = "<< vb.get_printable_value();  }  // print out the value

};  // Thats all!
```

Explanation of Introductory Example

The actual SNMP++ calls are made up of ten lines of code. A CTarget object is created using the IP address of the agent. A variable binding (Vb) object is then created using the object identifier of the MIB object to retrieve (System Descriptor). The Vb object is then attached to a Pdu object. An Snmp object is used to invoke an SNMP get. Once retrieved, the response message is printed out. All error handling code is included.

5. SNMP++ Features

5.1. Full Set of C++ based SNMP classes

SNMP++ is based around a set of C++ classes including the Object Identifier (Oid) class, Variable Binding (Vb) class, Protocol Data Unit (Pdu) class, Snmp class and a variety of classes making work with Abstract Syntax Notation (ASN.1) Structure of Management Information (SMI) types easy and object oriented.

5.2. Automatic SNMP Memory Management

The classes manage various SNMP structures and resources automatically when objects are instantiated and destroyed. This frees the application programmer from having to worry about de-allocating structures and resources and thus provides better protection from memory corruption and leaks. SNMP++ objects may be instantiated statically or dynamically. Static object instantiation allows destruction when the object goes out of scope. Dynamic allocation requires use of C++ constructs *new* and *delete*. Internal to SNMP++, are various Structure of Management Information (SMI) structures which are protected and hidden from the public interface. All SMI structures are managed internally, the programmer does not need to define or manage SMI structures or values. For the most part, usage of 'C' pointers in SNMP++ is non existent.

5.3. Ease Of Use

By hiding and managing all SMI structures and values, the SNMP++ classes are easy and safe to use. The programmer cannot corrupt what is hidden and protected from scope.

5.4. Power and Flexibility

SNMP++ provides power and flexibility which would otherwise be difficult to implement and manage. Each SNMP++ object communicates with an agent through a session model. That is, an instance of a SNMP++ session class maintains connections to specified agents. Each SNMP++ object provides reliability through automatic retry and time-outs. An application may have multiple SNMP++ object instances, each instance communicating to the same or different agent(s). This is a powerful feature which allows a network management application to have different sessions for each management component. Alternatively, a single Snmp session may be used for everything. For example, an application may have one SNMP++ object to provide graphing statistics, another SNMP++ object to monitor traps, and a third SNMP++ object to allow SNMP browsing. SNMP++ automatically handles multiple concurrent requests from different SNMP++ instances.

5.5. Portable Objects

The majority of SNMP++ is portable C++ code. Only the Snmp class implementation is different for each target operating system. ***If your program contains SNMP++ code, this code will port without any changes!***

5.6. Automatic Time-out And Retries

SNMP++ supports automatic time-out and retries. This frees the programmer from having to implement time-out or retry code. Retransmission policy is defined in the SnmpTarget class. This allows each managed target to have its own time-out / retry policy.

5.7. Blocked Mode Requests

SNMP++ includes a blocked model. The blocked mode for MS-Windows allows multiple blocked requests on separate SNMP class instances.

5.8. Non-Blocking Asynchronous Mode Requests

SNMP++ also supports a non-blocking asynchronous mode for requests. Time-outs and retries are supported in both blocked and asynchronous modes.

5.9. Notifications, Trap Reception and Sending

SNMP++ is designed to allow trap reception and sending on multiple transports including IP and IPX. In addition, SNMP++ allows trap reception and sending using non-standard trap IP ports and IPX socket numbers.

5.10. Support For SNMP Version 1 and 2 Through a Bilingual API

SNMP++ has been designed with support and usage for SNMP version one and two. All operations within the API are designed to be bilingual. That is, operations are not SNMP specific. Through utilization of the SnmpTarget class, SNMP version specific operations are abstracted.

5.11. SNMP Get, Get Next, Get Bulk, Set, Inform and Trap Supported

SNMP++ supports all six SNMP operations. All six SNMP++ member functions utilize similar parameter lists and operate in a blocked or non-blocked (asynchronous) manner.

5.12. Redefinition Through Inheritance

SNMP++ is implemented using C++ and thus allows a programmer to overload or redefine behavior which does not suite their needs. For example, if an application requires special Oid object needs, a subclass of the Oid class may be created, inheriting all the attributes and behavior the Oid base class while allowing new behavior and attributes to be added to the derived class.

6. SNMP++ for the Microsoft Windows Family of Operating Systems

SNMP++ has currently been implemented for MS-Windows 3.1, MS-Windows For Work Groups 3.11, MS-Windows NT 3.51, and MS-Windows '95.

6.1. Utilization of WinSNMP Version 1.1

WinSNMP version 1.1 is utilized to run SNMP++ on MS-Windows. This allows other SNMP applications coding directly to WinSNMP to coexist with SNMP++ applications. Note, the current HP implementation for MS-Windows utilizes WinSNMP. Other implementations are not required to use WinSNMP for ANS.1 encoding and decoding. Implementations which do not utilize WinSNMP are required to coexist with WinSNMP applications.

6.2. IP and IPX Support

For IP operation, a WinSock compliant stack is required. For IPX, a Novell Netware compatible client and the drivers are needed. SNMP++ has been tested to run over a wide variety of protocol stacks including FTP, Netmanage, LanWorkPlace, MS-WFWG 3.11, and Windows NT.

6.3. Notifications, Trap Receive and Send Support

SNMP++ includes support for interfacing with WinSNMP trap mechanisms. This includes both the ability to send traps and receive them. For the reception of traps, trap filtering capabilities are provided.

6.4. Compatibility with HP OpenView for Windows

A number of applications have been created using SNMP++ which coexist and are compatible with HP OpenView for MS-Windows.

7. SNMP++ for UNIX

7.1. Identical Class Interface

The class interface for the UNIX implementation is identical to SNMP++ for MS-Windows.

7.2. Portable to Windows-to-UNIX Emulators

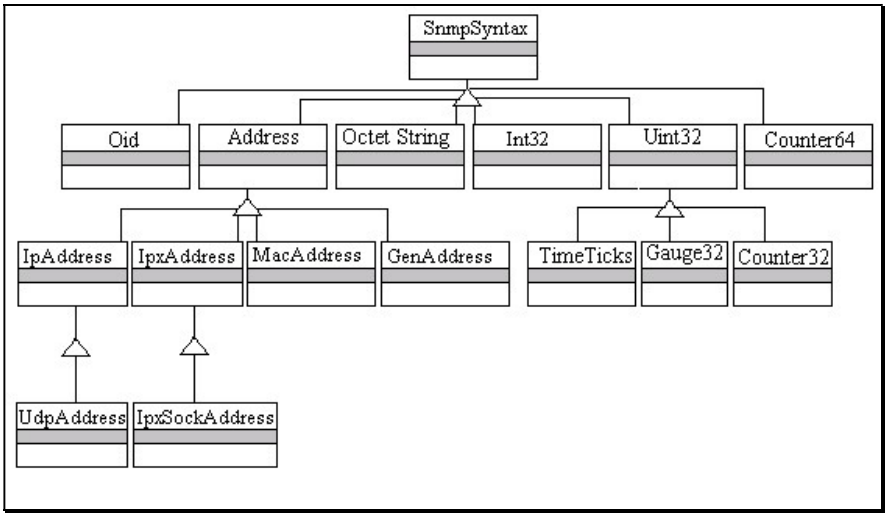
SNMP++ runs over UNIX by compiling and linking the proper SNMP++ class implementation. SNMP++ / UNIX is designed to run in a native text mode UNIX application, in a X-Window application , or using Windows-to-UNIX porting tools.

7.3. Compatibility with HP OpenView for UNIX

A number of applications have been created using SNMP++ which coexist and are compatible with HP OpenView for UNIX.

8. SNMP Syntax Classes

Object Modeling Technique (OMT) view of the SNMP++ SNMP Syntax Classes

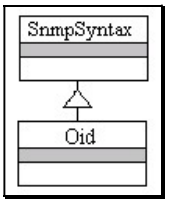


The SNMP++ SNMP syntax classes represent an object oriented C++ view of Structure of Management Information (SMI) Abstract Syntax Notation (ASN.1) data types which are used by SNMP. Included are a set of classes which map to the equivalent SMI types. In addition, a few non-SMI classes have been introduced for ease of use. SNMP++ gives these SNMP data types a powerful, easy to use interface. Below is a brief description of the various SNMP++ syntax classes.

SNMP++ Syntax Class Name	**Class Description**	**SMI or ASN.1 Counter part**
SnmpSyntax	Parent of all syntax classes.	No ASN.1 counter part, used for OO structure.
Oid	Object identifier class.	ASN.1 Object Identifier.
OctectStr	Octet string class.	ASN.1 Octet string.
Uint32	Unsigned 32 bit integer class.	SMI unsigned 32 bit integer.
TimeTicks	TimeTicks class.	SMI time ticks.
Counter32	32 bit counter class.	SMI 32 bit counter.
Gauge32	32 bit gauge class.	SMI 32 bit gauge.
Int32	Signed 32 bit integer.	SMI 32 bit signed integer.
Counter64	64 bit counter class.	SMI 64 bit counter.
Address	Abstract address class.	No ASN.1 counter part used for OO structure.
IpAddress	IP address class.	SMI IP address.
UdpAddress	UdpAddress class	SMI IP address with port specification.
IpxAddress	IPX address class.	No ASN.1 or SMI counter part
IpxSockAddress	IPX Address class with socket number.	No ASN.1 or SMI counter part
MacAddress	MAC address class.	SMI counter part
GenAddress	Generic Address	No ASN.1 or SMI counter part.

9. Object Id Class

Object Modeling Technique (OMT) view of the SNMP++ Oid Class



9.1. The Object Identifier Class

The Object Identification (Oid) class is the encapsulation of an SMI object identifier. The SMI object is a data identifier for a data element found in a Management Information Base (MIB), as defined by a MIB definition. The SMI Oid, its related structures and functions, are a natural fit for object orientation. In fact, the Oid class shares many common features to the C++ String class. For those of you familiar with the C++ String class or Microsoft's Foundation Classes (MFC) CString class, the Oid class will be familiar and easy to use. The Oid class is designed to be efficient and fast. The Oid class allows definition and manipulation of object identifiers. The Oid Class is fully portable and does not rely on any SNMP API to be present. The Oid class may be compiled and used with any ANSI C++ compiler.

9.2. Overview of Oid Class Member Functions

Oid Class Member Functions	**Description**
Constructors	
Oid::Oid(void);	Construct an empty Oid.
Oid::Oid(const char *dotted_string);	Construct an Oid with a dotted string.
Oid::Oid(const Oid &oid);	Construct an Oid with another Oid, copy constructor.
Oid::Oid(const unsigned long *data, int len);	Construct an Oid with a pointer and length.
Destructor	
Oid::~Oid( );	Destroy the Oid, frees up all memory held.

9.3. Overview of Oid Class Member Functions Continued

Oid Class Member Functions	Description
Overloaded Operators	
Oid & operator = (const char *dotted_string);	Assign an Oid a dotted string.
Oid & operator = (const Oid &oid);	Assign an Oid an Oid.
int operator == (const Oid &lhs, const Oid& rhs);	Compare two Oids for equivalence.
int operator == (const Oid& lhs, const char*dotted_string);	Compare an Oid and a dotted string for equivalence.
int operator != (const Oid &lhs, const Oid& rhs);	Compare two Oids for not equal.
int operator != (const Oid & lhs, const char *dotted_string);	Compare an Oid and dotted string for not equal.
int operator < (const Oid &lhs, const Oid& rhs);	Determine if one Oid is less than another Oid.
int operator < (const Oid &lhs, const char *dotted_string);	Determine if an Oid is less than a dotted string.
int operator <=(const Oid &lhs,const Oid &rhs);	Determine if one Oid is less than or equal to another Oid.
int operator <= (const Oid &lhs, const char *dotted_string);	Determine if one Oid is less than or equal to a dotted string.
int operator > (const Oid &lhs, const Oid &rhs);	Determine if one Oid is greater than another Oid.
int operator > (const Oid &lhs, const char * dotted_string);	Determine if one Oid is greater than a dotted string.
int operator >= (const Oid&lhs, const Oid &rhs);	Determine if one Oid is greater than or equal to another Oid.
int operator >= (const Oid &lhs, const char* dotted_string);	Determine if one Oid is greater than or equal to a dotted string.
Oid& operator += (const char *dotted_string);	Append a dotted string to an Oid.
Oid& operator +=(const unsigned long i);	Append a single value to a dotted string.
Oid& operator+=(const Oid& oid);	Append one Oid to another Oid.
unsigned long &operator [ ] (int position);	Access an individual sub-element of an Oid, read or write.
Output Member Functions	
char * get_printable(const unsigned int n);	Return the dotted format where n specifies how many sub elements to include.
char *get_printable(const unsigned long s, const unsigned long n);	Return the dotted format where s specifies the start position and n specifies how many sub elements to include.
char *get_printable();	Return the entire Oid as a dotted string.
operator char *();	Same as get_printable().
Miscellaneous Member Functions	
set_data (const unsigned long *data,const unsigned long n);	Set the data of an Oid using a pointer and a length.
unsigned long len( );	Return the length, number of sub elements, in an Oid.
trim(const unsigned long n=1);	Trim off the rightmost sub element of an Oid, default 1.
nCompare(const unsigned long n, const Oid& oid);	Compare the first n sub-ids (left to right) of an Oid parameter.
RnCompare(const unsigned long n, const Oid& oid);	Compare the last n sub-ids (right to left) of an Oid parameter.
int valid( );	Return the validity of an Oid.

9.4. Some Oid Class Examples

The following examples show different ways in which to use the Oid class. The Oid class does not require or depend on any other libraries or modules. The following code is ANSI/ISO C++ compatible.

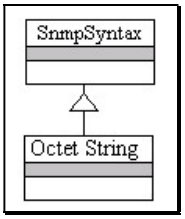
```
#include "oid.h"
void oid_example()
{
  // construct an Oid with a dotted string and print it out
  Oid o1("1.2.3.4.5.6.7.8.9.1");
  cout << "o1= " << o1.get_printable ();

  // construct an Oid with another Oid and print it out
  Oid o2(o1);
  cout << "o2=  " << o2.get_printable();

  // trim o2's last value and print it out
  o2.trim(1);
  cout <<  "o2=  " << o2.get_printable();

  // add a 2 value to the end of o2 and print it out
  o2+=2;
  cout << "o2=  " << o2.get_printable();

  // create a new Oid, o3
  Oid o3;

  // assign o3 a value and print it out
  o3="1.2.3.4.5.6.7.8.9.3";
  cout << "o3= " << o3.get_printable();

  // create o4
  Oid o4;

  // assign o4 o1's value
 o4=o1;

  // trim off o4 by 1
  o4.trim(1);

  // concat a 4 onto o4 and print it out
  o4+=".4";
  cout <<  "o4= " << o4.get_printable();

  // make o5 from o1 and print it out
  Oid o5(o1);
  cout << "o5= " << o5.get_printable();
```

Some Oid Class Examples Continued...

```
  // compare two not equal oids
  if (o1==o2)   cout << "O1 EQUALS O2";
  else    cout << "o1 not equal to o2";

  // print out a piece of o1
  cout << "strval(3) of O1 = " << o1.get_printable(3);

  // print out a piece of o1
  cout << "strval(1,3) of O1 = " << o1.get_printable(1,3);

  // set o1's last subid
  o1[ o1.len()-1] = 49;
  cout << "O1 modified = " << o1.get_printable();

  // set o1's 3rd subid
  o1[2]=49;
  cout << "O1 modified = " << o1.get_printable();

  // get the last subid of 02
  cout << "last of o2 = " << o2[o2.len()-1];

  // get the 3rd subid of 02
  cout << "3rd of o2 = " << o2[2];

  // ncompare
  if (o1.nCompare(3,o2))
    cout << "nCompare o1,o2,3 ==";
  else
   cout << "nCompare o1,o2,3 !=";

  // make an array of oids
  Oid oids[30]; int w;
  for ( w=0;w<30;w++)
  {
   oids[w] = "300.301.302.303.304.305.306.307";
   oids[w] += (w+1);
  }
  for (w=0;w<25;w++)
  {
   sprintf( msg,"Oids[%d] = %s",w, oids[w].get_printable());
   printf("%s",msg, strlen(msg));
   }
}
```

10. OctetStr Class

Object Modeling Technique (OMT) view of the SNMP++ Octet String Class



10.1. The OctetStr Class

The SNMP++ Octet class allows for easy and safe usage of SMI octets. With the octet class, it is no longer needed to work with octets using internal pointers and lengths. Using the SNMP++ OctetStr class, OctetStr objects can be easily instantiated, manipulated and destroyed without the overhead burden of managing memory and memory leaks. Like the ANSI C++ string class, the OctetStr class supports a variety of ways to construct OctetStr, assign them and use them with other SNMP++ classes. The OctetStr class interfaces with the SNMP++ variable binding (Vb) class making getting and setting the SMI value portion of a Vb object straight forward. The OctetStr class is fully portable and does not rely on additional SNMP libraries to be present.

10.2. Overview of OctetStr Class Member Functions

OctetStr Class Member Functions	**Description**
Constructors	
OctetStr::OctetStr(void);	Construct a OctetStr with no data.
OctetStr::OctetStr(const char* string);	Construct a OctetStr with a null terminated string.
OctetStr::OctetStr(const unsigned char *s, unsigned long int i);	Construct a OctetStr with a pointer and a length.
OctetStr::OctetStr(const OctetStr &octet);	Copy Constructor.
Destructor	
OctetStr::~OctetStr( );	Destroy an OctetStr object.

10.3. Overview of OctetStr Class Member Functions Continued

OctetStr Class Member Functions	Description
Overloaded Operators	
OctetStr& operator = (const char *string);	Assign an OctetStr object a null terminated string.
OctetStr& operator = (const OctetStr& octet);	Assign an OctetStr another OctetStr.
int operator == (const OctetStr &lhs, const OctetStr &rhs);	Compare two OctetStr objects for equivalence.
int operator == (const OctetStr & lhs, const char *string);	Compare an OctetStr and a char * for equivalence.
int operator != (const OctetStr& lhs, const OctetStr& rhs);	Compare two OctetStr objects for not equivalence.
int operator != (const OctetStr& lhs, const char *string);	Compare an OctetStr and a char * for not equivalence.
int operator < (const OctetStr&lhs, const OctetStr& rhs);	Test if one OctetStr is less than another.
int operator < (const OctetStr &lhs, const char * string);	Test if one OctetStr is less than a char *.
int operator <= (const OctetStr &lhs, const OctetStr &rhs);	Test if one OctetStr is less than or equal to another OctetStr.
int operator <= (const OctetStr &lhs, const char * string);	Test if one OctetStr is less than or equal to a char *.
int operator > (const OctetStr& lhs, const OctetStr &rhs);	Test if one OctetStr is greater than another OctetStr.
int operator > (const OctetStr& lhs, const char * string);	Test if one OctetStr is greater than a char *.
int operator >= (const OctetStr& lhs, const OctetStr &rhs);	Test if one OctetStr is greater than or equal to another OctetStr.
int operator >= (const OctetStr& lhs, const char *);	Test if one OctetStr is greater than or equal to a char *.
OctetStr& operator +=(const char * string);	Concatenate a string onto an OctetStr;
OctetStr& operator +=(const unsigned char c);	Concatenate a single char onto an OctetStr.
OctetStr& operator+=(const OctetStr &octetstr);	Concatenate a OctetStr object.
unsigned char& operator[ ] (int position i);	Allows array like access to an OctetStr.
Miscellaneous	
void set_data(const unsigned char *s, unsigned long l);	Set the data of an OctetStr using a pointer and length.
int nCompare(const unsigned long n, const OctetStr &o);	Compare n elements from parameter o.
unsigned long len();	Return the length of an OctetStr.
int valid();	Return the validity of an OctetStr.
unsigned char * data();	Returns pointer to internal data.
char * get_printable();	Formats for output, calls hex dump if not ASCII.
char * get_printable_hex();	Formats for output in hexadecimal format.

10.4. Special features

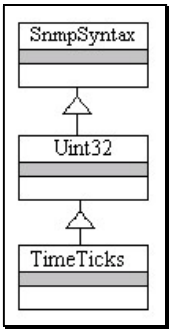
When printing out an OctetStr object, the char * or get_printable() member functions automatically invoke the get_printable_hex() member function if the octet string contains any character which is non ASCII. This allows the user to simply cast the OctetStr to a char * or fire the get_printable() member function and get nice output. The get_printable_hex() member function formats the OctetStr into a hexadecimal format.

10.5. Some OctetStr Class Examples

```
// Octet Class Examples
#include "octet.h"
void octet_example()
{
  OctetStr octet1;                              // create an invalid un- initialized octet object
  OctetStr octet2( "Red Hook Extra Bitter Ale"); // create an octet with a string
  OctetStr octet3( octet2);                     // create an octet with another octet
  unsigned char raw_data[50];                   // create some raw data
  OctetStr octet4( raw_data, 50);               // create an OctetStr using unsigned char data

  octet1 = "Sierra Nevada Pale Ale";            // assign one octet to another
  cout << octet1.get_printable();               // show octet1 as a null terminated string
  cout << octet4.get_printable_hex();           // show octet4 as a hex string
  cout << (char *) octet1;                      // same as get_printable()
  if ( octet1 == octet2)                        // compare two octets
    cout << "octet1 is equal to octet2";

  octet2 += "WinterFest Ale";                   // concat a string to an Octet
  if ( octet2 >= octet3)
    cout << "octet2 greater than or equal to octet2";

  octet2[4] = 'b';                              // modify an element of an OctetStr using [ ]'s

  cout << octet.len();                          // print out the length of an OctetStr

  unsigned char raw_data[100];
  octet1.set_data( raw_data, 100);              // set the data of an to unsigned char data

  cout << (octet1.valid() )? "Octet1 is valid" : "Octet1 is Invalid";  // get the validity of an OctetStr



}; // end octet example
```

11. TimeTicks Class

Object Modeling Technique (OMT) view of the SNMP++ TimeTicks Class



11.1. The TimeTicks Class

The SNMP++ TimeTicks provides benefits where SMI timeticks are needed. SMI timeticks are defined with the storage capabilities of an unsigned long integer. In addition to being an unsigned long int, SMI timeticks are treated as a distinct type. For this reason, the SNMP++ TimeTicks class has all the functionality and behavior of an unsigned long int, but is a separate class. Anything that can be done with an unsigned long integer can be done with a TimeTicks object. The TimeTicks class has additional behavior when interfacing with other SNMP++ classes like the Vb class. When used with the Vb class, TimeTicks objects can be set into (Vb::set) and gotten out of (Vb::get) of Vb objects. This allows the developer to get all the functionality of unsigned long and provide a one-to-one mapping to SMI timeticks.

11.2. Overview of TimeTicks Class Member Functions

TimeTicks Class Member Functions	Description
Constructors	
TimeTicks::TimeTicks(void);	Constructs an empty TimeTicks object.
TimeTicks::TimeTicks(const unsigned long i);	Construct a TimeTicks object using an unsigned long.
TimeTicks:;TimeTicks(const TimeTicks &t);	Construct a TimeTicks object using another TimeTicks object.
Destructor	
TimeTicks::~TimeTicks( );	Destroy a TimeTicks object.
Overloaded Operators	
TimeTicks& operator =(const TimeTicks &t);	Overloaded assignment operator.
char * get_printable();	Formats for output, in the form DD Days, HH:MM:SS.hh
operator unsigned long();	Gives unsigned long behavior to TimeTicks

11.3. Special Features

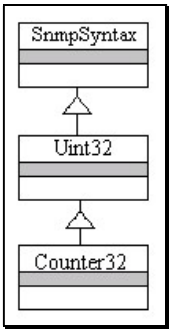
When printing out a TimeTicks object using TimeTicks::get_printable(), the value is formatted automatically to a “DD days, HH:MM:SS.hh” format where DD are the number of days, HH are the number of hours (24 hour clock), MM are the minutes, SS are the seconds and hh are the hundredths of a second.

11.4. Some TimeTicks Class Examples

```
// TimeTicks Examples
#include "timetick.h"
void timeticks_example()
{
   TimeTicks tt;                          // create an un-initialized timeticks instance
   TimeTicks tt1( (unsigned long) 57);    // create a timeticks using a number
   TimeTicks tt2( tt1);                   // create a timeticks using another instance

   tt = 192;                              // overloaded assignment to a number
   tt2 = tt;                              // overloaded assignment to another timeticks

   cout << tt.get_printable();            // print out in DD days, HH:MM:SS.hh
   cout << ( unsigned long) tt;           // print out unsigned long int value


}; // end timeticks example
```

12. Counter32 Class

Object Modeling Technique (OMT) view of the SNMP++ Counter32 Class



12.1. The Counter32 Class

The SNMP++ Counter class provides benefits where SMI 32 bit counters are needed. SMI counter are defined with the storage capabilities of an unsigned long integer. In addition to being an unsigned long integers, SMI counters are treated as a distinct type. For this reason, the SNMP++ Counter32 class has all the functionality and behavior of an unsigned long int, but is a separate class. Anything that can be done with an unsigned long int can be done with a Counter32 object. The Counter32 class has additional behavior when interfacing with other SNMP++ classes like the Vb class. When used with the Vb class, Counter32 objects can be set into (Vb::set) and gotten out of (Vb::get) of Vb objects. This allows the developer to get all the functionality of unsigned long and provide a one-to-one mapping to SMI counter.

12.2. Overview of Counter32 Class Member Functions

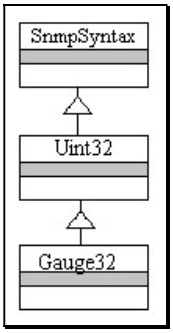
Counter32 Class Member Functions	Description
Constructors	
Counter32::Counter32(void);	Constructs an empty Counter32 object.
Counter32::Counter32(const unsigned long i);	Construct a Counter32 object using an unsigned long.
Counter32::Counter32(const Counter32 &c);	Construct a Counter32 object using another Counter32 object.
Destructor	
Counter32::~Counter32( );	Destroy a Counter32 object.
Overloaded Operators	
Counter32& operator = (const Counter32& c);	Overloaded assignment operator.
char * get_printable();	Returns Counter32 formatted for output.
operator unsigned long( );	Gives unsigned long behavior.

12.3. Some Counter32 Class Examples

```
// Counter Examples
#include "counter.h"
void counter_example()
{
   Counter32 ctr;                          // create an un-initialized counter instance
   Counter32 ctr1( (unsigned long) 57);    // create a counter using a number
   Counter32 ctr2(ctr1);                   // create a counter using another instance

   ctr = 192;                              // overloaded assignment to a number
   ctr1 = ctr;                             // overloaded assignment to another counter
   cout << (unsigned long) ctr;            // behave like an unsigned long int


}; // end counter example
```

13. Gauge32 Class

Object Modeling Technique (OMT) view of the SNMP++ Gauge32 Class



13.1. The Gauge32 Class

The SNMP++ Gauge32 class provides benefits where SMI 32 bit gauges are needed. SMI gauges are defined with the storage capabilities of an unsigned long integer. In addition to being an unsigned long int, SMI gauges are treated as a distinct type. For this reason, the SNMP++ Gauge32 class has all the functionality and behavior of an unsigned long integers but is a separate class. Anything that can be done with an unsigned long int can be done with a Gauge32 object. The Gauge32 class has additional behavior when interfacing with other SNMP++ classes like the Vb class. When used with the Vb class, Gauge32 objects can be set into (Vb::set) and gotten out of (Vb::get) of Vb objects. This allows the developer to get all the functionality of unsigned long and provide a one-to-one mapping to SMI gauge.

13.2. Overview of Gauge32 Class Member Functions

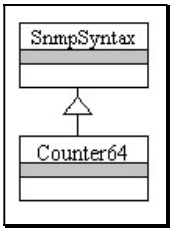
Gauge32 Class Member Functions	**Description**
Constructors	
Gauge32::Gauge32(void);	Constructs an empty Gauge32 object.
Gauge32::Gauge32(const unsigned long i);	Construct a Gauge32 object using an unsigned long.
Gauge32::Gauge32(const Gauge32 &g);	Construct a Gauge32 object using another Gaueg32 object.
Destructor	
Gauge32::Gauge32( );	Destroy a Gauge32 object.
Overloaded Operators	
Gauge32& operator = (const Gauge32 &g);	Overloaded assignment operator.
char * get_printable();	Returns formatted Gauge32 for output.
operator unsigned long( );	Gives unsigned long behavior.

13.3. Some Gauge32 Examples

```
// Gauge Examples
#include "gauge.h"
void gauge_example()
{
   Gauge32 gge ;                          // create an un-initialized Gauge instance
   Gauge32gge1( (unsigned long) 57);      // create a Gauge using a number
   Gauge32 ctr2(ctr1);                    // create a Gauge using another instance

   gge = 192;                             // overloaded assignment to a number
   gge1 = gge;                            // overloaded assignment to another counter
   cout << (unsigned long) gge;           // behave like an unsigned long int


}; // end gauge example
```

14. Counter64 Class

Object Modeling Technique (OMT) view of the SNMP++ Counter64 Class



14.1. The Counter64 Class

The SNMP++ 64bit counter class allows for the usage of SMI 64 bit counters. 64 bit counters are defined as a SNMP version 2 SMI variable. So, for SNMP version 1, this MIB variable does not exist. The Counter64 class allows for easy usage of 64 bit counters which are made up of two unsigned long portions (high and low). The Counter64 class provides overloaded operators for addition, subtraction, multiplication and division, giving the Counter64 class a natural feel.

14.2. Overview of Counter64 Class Member Functions

Counter64 Class Member Functions	**Description**
Constructors	
Counter64::Counter64(void);	Construct a Counter64 with no data.
Counter64::Counter64(const unsigned long hi, const unsigned long low);	Construct a Counter64 with two unsigned long ints.
Counter64::Counter64(const Counter64 &ctr64);	Copy Constructor.
Counter64::Counter64(const unsigned long ul);	Construct a Counter64 with a single unsigned long.
Destructor	Destroy an OctetStr object
Counter64::~Counter64( );	

14.3. Overview of Counter64Class Member Functions Continued

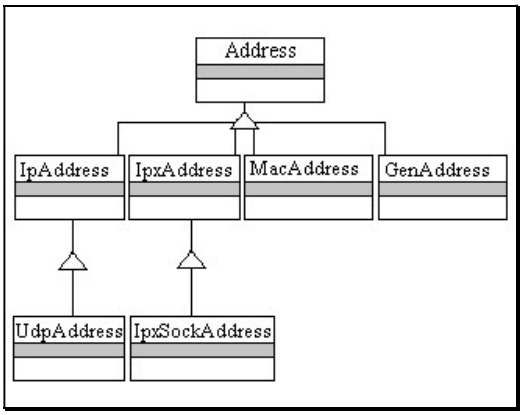
Counter64 Class Member Functions	Description
Overloaded Operators	
Counter64& operator = (const Counter64 &ctr64);	Assign a Counter64 to a Counter64.
Counter64& operator = (const unsigned long i);	Assign a Counter64 an unsigned long, sets low, clears high.
Counter64 operator + (const Counter64 &ctr64);	Add two Counter64's.
Counter64 operator - (const Counter64 &ctr64);	Subtract two Counter64's.
Counter64 operator * (const Counter64 &ctr64);	Multiply two Counter64's.
Counter64 operator / (const Counter64 &ctr64);	Divide two Counter64's.
int operator == (Counter64 &lhs, Counter64 &rhs);	Test if two Counter64's are equal.
int operator != (Counter64 &lhs, Counter64 &rhs);	Test if two Counter64's are not equal.
int operator < (Counter64 &lhs, Counter64 &rhs);	Test if one Counter64 is less than another Counter64.
int operator <= (Counter64 &lhs, Counter64 &rhs);	Test if one Counter64 is less or equal to than another Counter64.
int operator > (Counter64 &lhs, Counter64 &rhs);	Test if one Counter64 is greater than another Counter64
int operator >= (Counter64 &lhs, Counter64 &rhs);	Test if one Counter64 is greater than or equal to than another Counter64.
Member Functions	
unsigned long high();	Returns high portion.
unsigned long low();	Returns low portion.
void set_high();	Sets the high portion.
void set_low();	Sets the low portion.

14.4. Some Counter64 Class Examples

```
// Counter64 examples
#include "ctr64.h"
void counter64_example()
{
  Counter64 c64;                  // instantiate a 64 bit counter object with no parms
  Counter64 my_c64( 100, 100);    // instantiate a 64 bit counter with a hi and low value
  Counter64 your_c64( my_c64); // instantiate a 64 counter using another 64bit counter

  cout << my_c64.high();          // print out the high portion of the c64
  cout << my_c64.low();           // print out the low portion of the c64

  c64 = my_c64 + your_c64;        // overloaded addition
  c64 = my_c64 * your_c64;        // overloaded multiplication
  c64 = my_c64 / your_c64;        // overloaded division
  c64 = my_c64 - your_c64;        // overloaded subtraction

   if ( c64 == my_c64)            // overloaded equivalence test
     cout << "c64 equals my_c64\n";

   if ( c64 != my_c64)            // overloaded not equal test
     cout << "c64 not equal to my_c64\n";

  if ( c64 < my_c64)              // overloaded less than
     cout << "c64 less than my_c64\n";

}; // end Counter64 example
```

15. Address Class

Object Modeling Technique (OMT) view of the SNMP++ Address Class



15.1. What is the Network Address Class?

The network address class is a set of C++ classes which provide for simple, safe, portable and efficient use of network addresses. Most network management applications require use of network addresses for accessing and managing devices. This includes address validation, modification and user interface control. Rather than manage all the internal details of particular network addresses, the Address class encapsulates and hides the internal mechanisms freeing the application programmer to focus on the problem at hand. The motivation for the development of the Address class emerged from input and discussion at the '95 Interop SNMP++ Birds-of-A-Feather (BOF) and from dialog with Hewlett-Packard OpenView programmers.

15.2. Why use the Network Address Class?

The address class provides a number of benefits including: automatic memory management, address validation, portability to any C++ environment, ease of use and extensibility. Currently, the Address class consists of four classes, the IpAddress Class, the IpxAddress Class, the MacAddress class and the GenAddress class. In the future other subclasses will be added including IP Next Generation (IPng).

15.3. Address Classes

The address classes are based around one abstract class, the Address class. This is an abstract class. That is, there may be no instances of this class. The Address class provides a consistent interface through the use of virtual member functions. This allows passing addresses to other functions using the generic interface. This minimizes code changes to modules using addresses.

15.4. Address Classes and Interfaces

The base class, the Address class is an abstract class. The class contains the commonality of all derived address classes. This includes an identical interface for constructing, accessing and mutating Addresses.

Address Class Member Functions	Description
IpAddress Class Constructors	
IpAddress::IpAddress(void);	Construct an empty IpAddress object.
IpAddress::IpAddress(const char *string);	Construct an IpAddress from a string, do DNS if needed.
IpAddress::IpAddress(const IpAddress &ipa);	Copy constructor.
IPAddress Member Functions	
char * friendly_name(int & status);	Invokes DNS lookup for friendly name.
UdpAddress Constructors	
UdpAddress::UdpAddress(void);	Construct an invalid UdpAddress Object.
UdpAddress::UdpAddress(const char *string);	Construct a UdpAddress using a char string.
UdpAddress::UdpAddress(const UdpAddress &udpa);	Construct a UdpAddress using another UdpAddress.
UdpAddress Member Functions	
void UdpAddress:set_port(const unsigned int p);	Set the port number for a UdpAddress object.
unsigned int UdpAddress::get_port();	Get the port number from a UdpAddress object.
IpxAddress Class Constructors	
IpxAddress::IpxAddress(void);	Construct an empty IPX address.
IpxAddress::IpxAddress(const char *string);	Construct an IPX address using a char string.
IpxAddress::IpxAddress(const IpxAddress &ipxa);	Copy constructor.
IpxSockAddress Constuctors	
IpxSockAddress::IpxSockAddress(void);	Construct an empty IpxSockAddress object.
IpxSockAddress::IpxSockAddress(const char *string);	Construct a IpxSockAddress using a char string.
IpxSockAddress::IpxSockAddress(const IpxSockAddress &ipxs);	Construct a IpxSockAddress using another IpxSockAddress.
IpxSockAddress Member Functions	
IpxSockAddress::set_socket(const unsigned int s);	Get the socket number from a IpxSockAddress.
unsigned int IpxSocketAddress::get_socket();	Set the socket number into a IpxSockAddress.
MacAddress Constructors	
MacAddress::MacAddress(void);	Construct an empty MacAddress object.
MacAddress::MacAddress(const char * string);	Construct a MacAddress from a string.
MacAddress::MacAddress(const MacAddress &mac);	Copy constructor.
GenAddress Constructors	
GenAddress::GenAddress(void);	Construct a invalid GenAddress object.
GenAddress::GenAddress(const char * addr);	Construct a GenAddress using a string.
GenAddress::GenAddress(const GenAddress &addr);	Copy constructor.
Common Member Functions, applicable to all Address classes	
int operator == (const Address &lhs, const Address rhs);	Determine if two Addresses are equal.
int operator != (const Address &lhs, const Address &rhs);	Determine if two Addresses are not equal.
int operator > (const Address &lhs, const Address &rhs);	Determine if one Address is greater than another.
int operator >= (const Address &lhs, const Address &rhs);	Determine if one Address is greater than or equal.
int operator < (const Address &lhs, const Address &rhs);	Determine if one Address is less than another.
int operator<=(const Address &lhs, const Address &rhs);	Determine if one Address is less than or equal to another.
int operator == (const Address &lhs, cosnt char *inaddr);	Determine if two Addresses are equal.
int operator > (const Address &lhs, const char *inaddr);	Determine if an Address if greater than a string.
int operator < (const Address &lhs, const char *inaddr);	Determine if an Address is less than a string.
virtual int valid( );	Determine if an Address is valid.
unsigned char& operator[](int position);	Allow access to an Address object using array like access.
char * get_printable ( );	Returns Address formatted for output.

15.5. IpAddress Class Special Features

The IpAddress class will do automatic Domain Name Services (DNS) lookup when calling the Address::get_printable() member function. If the DNS is not active or if the address

cannot be resolved, the dotted format will be returned. Alternatively, an IpAddress can be constructed with a friendly name. In this case the constructor will invoke the DNS lookup. If the friendly name cannot be found, the address is invalid. This powerful feature allows you to utilize friendly names in your IpAddress user presentation.

15.6. GenAddress

The GenAddress class allows creation and usage of generic addresses where a GenAddress may take on the behavior and attributes of any of the other Address classes (IpAddress, IpxAddress and MacAddress). When working with arbitrary addresses, you may use a GenAddress. The constructor for the GenAddress class allows creating an Address with any character string. The constructor determines the specific type of Address which matches the string and thereafter gives the GenAddress the attributes and behavior of that Address. This saves the programmer from having to write code which explicitly deals with the differences across Addresses.

GenAddress Examples

```
GenAddress address1("10.4.8.5");                    // make an IP GenAddress
GenAddress address2("01020304-10111213141516");     // make an IPX GenAddress
GenAddress address3("01:02:03:04:05:06");           // make a MAC GenAddress

cout << address3.get_printable();                   // print out the GenAddress

if ( !address1.valid())                             // check validity
   cout << "address1 ! valid";
```

15.7. Address Class Validation

All address classes support the ::valid() member function. The ::valid() member function returns the validity of the particular address object. Validation is determined when constructing or assigning address objects. After assignment, the ::valid() member function may be used to determine validity.

Address Class Validation Examples

```
MacAddress mac;
mac = "01.010a0d";                                  // invalid MAC address
printf("%s", (mac.valid() ? "Valid" :"Invalid"));
```

15.8. UdpAddresses and IpxSockAddresses

For most usage, users of SNMP++ will utilize the well know port and socket numbers for SNMP operations. For the Internet Protocol (IP) this includes using port 161 for an agent's destination port and port 162 for the trap / notification reception port. There are time when alternate port / socket specification is required. For these instances, the UdpAddress class and IpxSockAddress class allow definition of port or socket information.

15.8.1. Using UdpAddresses for Making Requests

When requesting information from an agent which does not listen on the standard well know port, the UdpAddress class should be used. The UdpAddress class supports two member functions for setting and getting custom port information. Attaching a UdpAddress to a Target and using it for requests will cause SNMP++ to utilize the custom port number.

15.8.2. Using IpxSockAddresses for Making Requests

When requesting information from an agent which does not listen on the standard well know IPX socket number, the IpxSockAddress class should be used. The IpxSockAddress class supports two member functions for setting and getting custom socket number information. Attaching a IpxSockAddress to a Target and using it for requests will cause SNMP++ to utilize the custom socket number.

15.8.3. Using UdpAddress and IpxSockAddress for Notification Reception

UdpAddresses and IpxSockAddresses may also be used to specify alternate ports and sockets for notification reception. This allows applications to receive traps and informs on non standard ports and socket numbers.

15.9. Valid Address Formats

Valid Address formats for addresses are currently defined as:

Valid IP format BNF Grammar XXX.XXX.XXX.XXX
ip-address : ip-token DOT ip-token DOT ip-token DOT ip-token
DOT : '.'
ip-token : [0-255]

Valid IPX format BNF GrammarXXXXXXXX:XXXXXXXXXXXX
ipx-address: net-id SEPARATOR mac-id
SEPARATOR : ' ' | ':' | '-' | '.'
net_id : 1{byte-token}4
mac-id: 1{byte-token}6
byte-token: 1{byte}2
byte: [0-9|a-f|A-F]

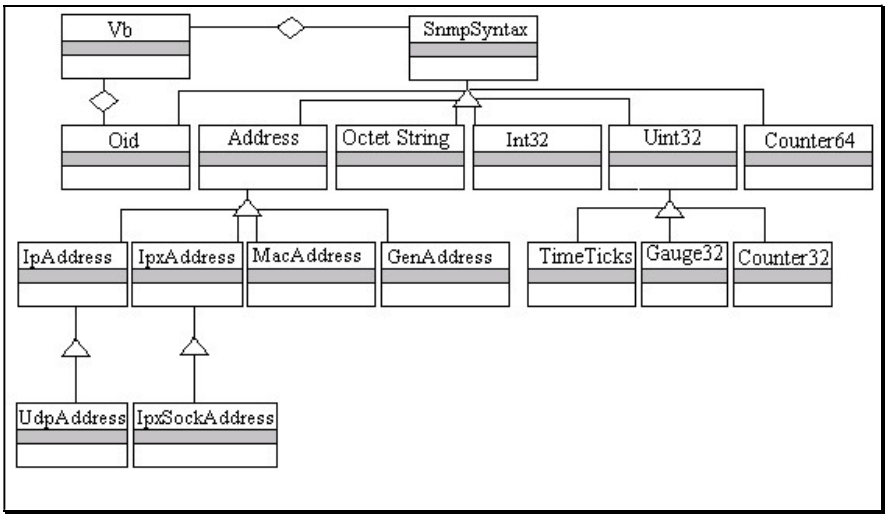
Valid MAC format BNF Grammar XX:XX:XX:XX:XX:XX
mac-id: byte_token colon byte_token colon byte_token colon byte_token colon byte_token
byte-token: 1{byte}2
byte: [0-9|a-f|A-F]
colon: ':'

15.10. Address Class Examples

```
// address class examples
#include "address.h"
void address_examples()
{
  //--------------[ IPAddress construction ]---------------------------------------------------
  IpAddress ip1();                        // makes an invalid IpAddress object
  IpAddress ip2("10.4.8.5");              // makes a IpAddress verifies dotted format
  IpAddress ip3(ip2);                     // makes an IpAddress using another IpAddress
  IpAddress ip4("trout.rose.hp.com");     // makes an IpAddress does DNS on string

  //-------------[ IPX Address construction ]--------------------------------------------------
  IpxAddress ipx1();                              // makes an invalid IPX address
  IpxAddress ipx2(");                             // makes and verifies an IPX address
  IpxAddress ipx3( ipx2);                         // makes an IPX from another IPX

  //--------------[ MAC Address construction ]--------------------------------------------------
  MacAddress mac1();                              // makes an invalid MAC address
  MacAddress mac2("08:09:12:34:52:12");           // makes and verifies a MAC address
  MacAddress mac3( mac2);                         // makes a MAC from another MAC

 //---------------[ Gen Address Construction ]-------------------------------------------------
 GenAddress addr1("10.4.8.5");
 GenAddress addr2("01020304:050607080900");

  //--------------[ printing addresses ]-------------------------------------------------------
  cout << (char *) ip2;
  cout << (char *) ipx2;
  cout << (char *) mac2;

  //---------------[ assigning Addresses ]-----------------------------------------------------
  ip1 = "15.29.33.10";
  ipx1 = "00000001-080912345212";
  mac1 = "08:09:12:34:52:12";

  //--------------[ comparing Addresses ]------------------------------------------------------
  if ( ip1 == ip2)
     cout << "ip1 == ip2";

  if (ipx1 != ipx2)
    cout << "ipx1 != ipx2";

  if ( mac1 <= mac2)
    cout << "mac1 < mac2";

//---------------[ modifying an address ]-----------------------------------------------------
  mac1[4] = 15;
  cout << mac2[2];


}; // end address examples
```

16. The Variable Binding Class

Object Modeling Technique (OMT) view of SNMP++ Variable Binding (Vb) Class



The variable binding (Vb) class represents the encapsulation of a SNMP variable binding. A variable binding is the association of a SNMP object ID with its SMI value. In object oriented methodology, this is simply a *has a* relation. A Vb object *has an* Oid object and a SMI value. The Vb class allows the application programmer to instantiate Vb objects and assign the Oid portion (Vb::set_oid), and assign the value portion (Vb::set_value). Conversely, the Oid and value portions may be extracted using Vb::get_oid() and Vb::get_value() member functions. The public member functions Vb::set_value() and Vb::get_value() are overloaded to provide the ability to set or get different SMI values to the Vb binding. Variable binding lists in SNMP++ are represented as arrays of Vb objects. All SMI types are accommodated within the Vb Class. The Vb class provides full data hiding. The user does not need to know about SMI value types, Oid internal representations, or other related SNMP structures. The Vb class is fully portable using a standard ANSI C++ compiler.

16.1. Variable Binding Class Member Functions Overview

Variable Binding Class Member Functions	Description
Constructors	
Vb(void);	Construct an empty Vb object.
Vb(const Oid &oid);	Construct a Vb with an Oid portion.
Vb(const Vb &vb);	Copy Constructor.
Destructor	
~Vb();	Destroy a Vb, free up all resources.
Set Oid / Get Oid	
void set_oid(const Oid &oid);	Set the Oid portion of a Vb.
void get_oid(Oid &oid) const;	Get the Oid portion.
Set Value	
void set_value(const SMIValue &val);	Set the value to any other SmiValue.
void set_value(const int i);	Set the value to an integer.
void set_value(const long int i);	Set the value to a long integer.
void set_value(const unsigned long int i);	Set the value to an unsigned long integer.
void set_value(const char WINFAR * ptr);	Set the value to a null terminated string.
Get Value	
int get_value(SMIValue &val);	Get the value, use any SmiValue.
int get_value(int &i);	Get an integer value.
int get_value(long int &i);	Get an signed long integer.
int get_value(unsigned long int &i);	Get an unsigned long integer.
int get_value(unsigned char WINFAR * ptr, unsigned long &len);	Get an unsigned char array, returns data and a len.
int get_value(unsigned char WINFAR * ptr, unsigned long &len, unsigned long maxlen);	Get a unsigned char array and a len, up to max len in size.
int get_value(char WINFAR *ptr);	Get a null terminated string.
Miscellaneous	
SmiUINT32 get_syntax();	Returns SMI syntax.
char *get_printable_value();	Returns formatted value.
char *get_printable_oid();	Returns formatted Oid portion.
void set_null();	Sets a Vb object to hold a null value.
int valid();	Returns validity of a Vb.
Overloaded Operators	
Vb& operator=(const Vb &vb);	Assign one Vb to another.

16.2. Vb Class Public Member Functions

The Vb class provides a variety of public member methods to access and modify Vb objects.

```
// A Vb object may be constructed with no arguments. In this case, the Oid and
// value portions must be set with subsequent member function calls.
// constructor with no arguments
// makes an Vb, un-initialized
Vb::Vb( void);
```

16.2.1. Vb Class Constructors & Destructors

Alternatively, a Vb object may be constructed with an Oid object as a construction parameter. This initializes the Oid part of the Vb object to the Oid passed in. The Vb object makes a copy of the Oid passed in. This saves the programmer from having to worry about the duration of the parameter Oid.

```
// constructor to initialize the Oid
// makes a Vb with Oid portion initialized
Vb::Vb( const Oid oid);
```

The destructor for a Vb object releases any memory and/or resources which were occupied. For statically defined objects, the destructor is called automatically when the object goes out of scope. Dynamically instantiated objects require usage of the *delete* construct to cause destruction.

```
// destructor
// if the Vb has a Oid or an octet string then
// the associated memory needs to be freed
Vb::~Vb();
```

16.2.2. Vb Class Get Oid / Set Oid Member Functions

The get and set Oid member functions allow getting or setting the Oid part of a Vb object. When doing SNMP gets or sets, the variable is identified by setting the Oid value of the Vb via the Vb::set_oid(Oid oid). Conversely, the Oid portion may be extracted via the Vb::get_oid(Oid &oid) member function. The get_oid member function is particularly useful when doing SNMP get next.

The Oid portion of a Vb object can be set with an already constructed Oid object

```
// set value Oid only with another Oid
void Vb::set_oid( const Oid &oid);
```

The Oid portion may be retrieved by providing a target Oid object. This destroys the previous value of the Oid object.

```
// get Oid portion
void Vb::get_oid( Oid &oid);
```

16.2.3. Vb Class Get Value / Set Value Member Functions

The get_value, set_value member functions allow getting or setting the value portion of a Vb object. These member functions are overloaded to provide getting or setting different types. The internal hidden mechanisms of getting or setting Vb's handles all memory allocation/de-allocation. This frees the programmer from having to worry about SMI-value structures and their management. Get value member functions are typically used to get the value of a Vb object after having done a SNMP get. Set value member functions are useful when wishing to set values of Vb's when doing a SNMP set. The get_value member functions return a -1 if the get does not match what the Vb is holding.

Set the value portion of a Vb object to an integer. This maps to an SMI INT.

```
// set the value with an int
void Vb::set_value( const int i);
```

Set the value portion of a Vb Object to a long integer. This maps to an SMI INT32.

```
// set the value with a long signed int
void Vb::set_value( const long int i);
```

Set the value portion of a Vb object to an unsigned long integer. This maps to an SMI UNIT32.

```
// set the value with an unsigned long int
void Vb::set_value( const unsigned long int i);
```

Set the value portion of a Vb to Gauge32 object. This maps to an SMI 32 bit gauge.

```
// set the value with a 32 bit gauge
void Vb::set_value( const Gauge32 gauge);
```

Set the value portion of a Vb object to a TimeTicks object. This maps to an SMI time ticks variable.

```
// set the value with a TimeTicks
void Vb::set_value( const TimeTicks timeticks);
```

Set the value portion of a Vb object to a Counter32 object. This maps to an SMI 32 bit counter.

```
// set value with a 32 bit counter
void Vb::set_value( const Counter32 counter);
```

Set the value portion of a Vb object to a Counter64 object. This is used for SMI 64 bit counters comprised of a hi and low 32 bit portion.

```
// set value to a 64 bit counter
void Vb::set_value( const Counter64 c64);
```

Set the value portion of a Vb object to an Oid.

```
// set value for setting an Oid
// creates own space for an Oid which
// needs to be freed when destroyed
void Vb::set_value( const Oid &varoid);
```

Set the value portion of a Vb object to a char string. Really, this internally uses the SMI value portion of an octet string but makes it easier to use when it is an ASCII string. (eg system descriptor)

```
// set value on a string
// makes the string an octet
// this must be a null terminates string
void Vb::set_value( const char  * ptr);
```

Set the value portion of a Vb to an IP address object. This member function utilizes the Address class. IP address is a explicit SMI value type.

```
// set an IPAddress object as a value
void Vb::set_value ( const IpAddress ipaddr);
```

Set the value portion of a Vb to an IPX address object. This member function utilizes the Address class. IPX address is treated as an octet SMI value type.

```
// set an IPXaddress object as a value
void Vb::set_value ( const IpxAddress ipxaddr);
```

Set the value portion of a Vb to an MAC address object. This member function utilizes the Address class. MAC address is treated as an octet SMI value type.

```
// set an MAC address object as a value
void Vb::set_value ( const MacAddress macaddr);
```

16.2.4. Set the value to a GenAdress object.

```
// set an GenAddress object as a value
void Vb::set_value ( const GenAddress genaddr);
```

16.2.5. Set the value to a UdpAdress object.

```
// set an UdpAddress object as a value
void Vb::set_value ( const UdpAddress udpaddr);
```

```
// set an IpxSockAddress object as a value
void Vb::set_value ( const IpxSockAddress ipxsockaddr);
```

16.2.6. Set the value to a IpxSockAdress object.

16.2.7. Set the value portion of a Vb to an Octet object.

```
// set the value portion to a SNMP++ Octet object
void Vb::set_value( const OctetStr octet);
```

16.2.8. Vb Class Get Value Member Functions

All Vb::get_value member functions modify the parameter passed in. If a Vb object does not contain the requested parameter type, the parameter will not be modified and a SNMP_CLASS_INVALID will be returned. Otherwise on success, a SNMP_CLASS_SUCCESS status is returned.

Get an integer value from a Vb object.

```
// get value int
// returns 0 on success and value
int Vb::get_value( int &i);
```

Get a long integer from a Vb object.

```
// get the signed long int
int Vb::get_value( long int &i);
```

Get an unsigned long integer value from a Vb.

```
// get the unsigned long int
int Vb::get_value( unsigned long int &i);
```

Get a Gauge32 from a Vb object.

```
// get a Gauge32
int Vb::get_value( Gauge32 &gauge);
```

Get a TimeTicks from a Vb object.

```
// get a TimeTicks from a Vb
int Vb:get_value( TimeTicks &timeticks);
```

Get a Counter32 from a Vb object.

```
// get a counter from a Vb
int Vb::get_value(Counter32 &counter);
```

Get a 64 bit counter from a Vb object.

```
// get a 64 bit counter
int Vb::get_value( Counter64 &counter64);
```

Get an Oid object from a Vb object.

```
// get the Oid value
// free the existing Oid value
// copy in the new Oid value
int Vb::get_value( Oid &varoid);
```

Get an unsigned char string value from a Vb object (Octet string).

```
// get a unsigned char string value
// destructive, copies into given ptr of up
// to len length
int Vb::get_value( unsigned char  * ptr, unsigned long &len);
```

Get a char string from a Vb object. This grabs the octet string portion and pads it with a null.

```
// get a char * from an octet string
// the user must provide space or
// memory will be stepped on
int Vb::get_value( char  *ptr);
```

Get a IP address Object from a Vb object. IpAddress is defined as an Address object.

```
// get an IPAddress
int Vb::get_value( IpAddress &ipaddr);
```

Get a IPX address Object from a Vb object. IpxAddress is defined as an Address object.

```
// get an IPXAddress
int Vb::get_value( IpxAddress &ipxaddr);
```

Get a MAC address Object from a Vb object. MacAddress is defined as an Address object.

```
// get an MAC address
int Vb::get_value( MacAddress &MACaddr);
```

Get a GenAddress Object from a Vb object. GenAddress is defined as an Address object.

```
// get an gen address
int Vb::get_value( GenAddress &genaddr);
```

Get a UdpAddress Object from a Vb object. UdpAddress is defined as an Address object.

```
// get an Udp address
int Vb::get_value( UdpAddress &Udpaddr);
```

Get a IpxSockAddress Object from a Vb object. IpxSockAddress is defined as an Address object.

```
// get an IpxSockAddress
int Vb::get_value( IpxSockAddress &IpxSockAddr);
```

Get an Octet Object from a Vb Object.

```
// get an Octet object from a Vb
int Vb::get_value( OctetStr, &octet);
```

16.2.9. Vb Object Get Syntax Member Function

This method violates the object oriented paradigm. An object knows what it is. By having a method which returns the id of an object violates its data hiding. Putting that aside, there are times when it may be necessary to know what value a Vb is holding to allow extracting that value. For example, when implementing a browser it would be necessary to grab a Vb, ask it what it has and then pull out whatever it may hold. Returned syntax values are SMI syntax value.

```
// return the current syntax
// This method violates the OO paradigm but may be useful if
// the caller has a Vb object and does not know what it is.
// This would be useful in the implementation of a browser.
SmiUINT32 get_syntax();
```

16.2.10.Vb Object Validation Check

An instantiated Vb object may be checked to determine if it is valid by invoking the Vb::valid() member functions. Valid Vbs are those which have been assigned an Oid.

```
// determine if a Vb object is valid
int Vb::valid();
```

16.2.11.Vb Object Assignment to Other Vb Objects

Vb objects may be assigned to one another using the overloaded assignment operator, =. This allows for easy assignment of one Vb object to another without having to interrogate a Vb object for its contents and then assign them manually to the target Vb object.

```
// overloaded Vb assignment
// assignment to another Vb object overloaded
Vb& operator=( const &Vb vb);
```

16.2.12.Vb Object Errors

When getting data from a variable binding object, Vb::get_value(), an error can occur based on the value the Vb has and the type of value you are requesting. For example, assume a Vb object has an OctetStr and you are attempting to get a TimeTicks object out of it. The Vb::get_value() will fail since a TimeTicks object cannot be returned. In the event an error occurs, the caller mat utilize Vb::get_syntax() to interrogate the Vb for its actual value or exception value.

Vb::get_value() return value	Description
SNMP_CLASS_SUCCESS	Success, requested value was returned.
SNMP_CLASS_INVALID	Error, Vb value does not hold requested value.

16.3. Vb Class Examples

The following examples show different ways in which to use the Vb class. The Vb class does not require or depend on any other libraries or modules other than the Oid class. The following C++ code is ANSI compatible.

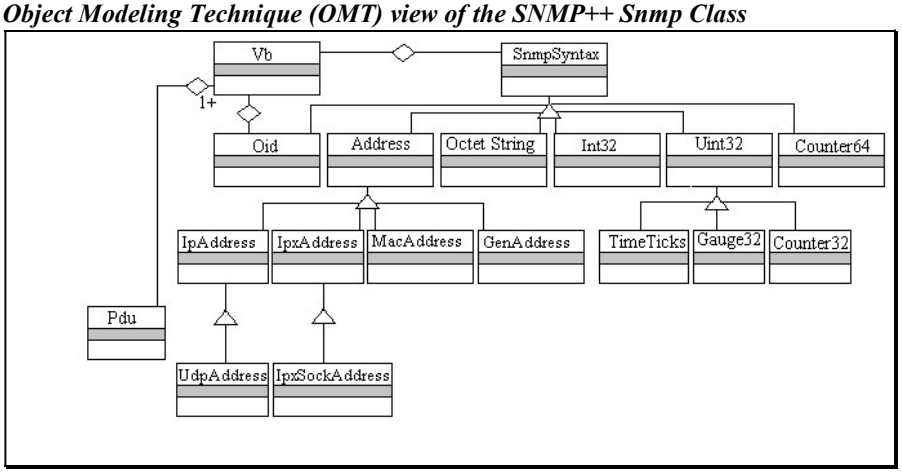
```
#include "oid.h"
#include "vb.h"
vb_test()
{
  // -------[Ways to construct Vb objects ]-------
  // construct a single Vb object
  Vb vb1;

  // construct a Vb object with an Oid object
  // this sets the Oid portion of the Vb
  Oid d1("1.3.6.1.4.12");
  Vb vb2(d1);

  // construct a Vb object with a dotted string
  Vb vb3( (Oid) "1.2.3.4.5.6");

  // construct an array of ten Vbs
  Vb vbs[10];

  //------[Ways to set and get the Oid portion of Vb objects ]

  // set and get the Oid portion
  Oid d2((Oid)"1.2.3.4.5.6");
  vb1.set_oid(d2);
  Oid d3;
  vb1.get_oid(d3);
  if (d2==d3)  cout << "They better be equal!!\n";

  Vb ten_vbs[10];
  int z;
  for (z=0;z<10;z++)
  ten_vbs[0].set_oid((Oid)"1.2.3.4.5");

  //-------[ ways to set and get values ]

  // set & get ints
  int x,y;
  x=5;
  vb1.set_value(x);
  vb1.get_value(y);
  if ( x == y) cout << "x equals y\n";
  // set and get long ints
  long int a,b;
  a=100;
```

Vb Class Example Continued..

```
//-------[ ways to set and get values ]
 if ( a == b) cout << "a equals b\n";
 // set & get unsigned long ints
unsigned long int c,d;
c = 1000;

 vbs[0].set_value( c);   vbs[0].get_value( d);
 if ( c == d) cout << "c equals d\n";

 // set a 64 bit counter
 Counter64 c64(1000,1001);
 vbs[1].set_value( c64);

 // get and set an oid as a value
 Oid o1, o2;
 o1 = "1.2.3.4.5.6";
 vbs[2].set_value( o1);   vbs[2].get_value( o2);
 if ( o1 == o2) cout << "o1 equals o2\n";

 // set and get an octet string
 unsigned char data[4],outdata[4];
 unsigned long len,outlen;
 len =4;  data[0] = 10; data[1] = 12; data[2] = 12; data[3] = 13;
 OctetStr octetstr(data,len);
 vbs[3].set_value( octetstr);
 vbs[3].get_value( octetstr);

 // get & set a string
 char beer[80];   char good_beer[80];
 strcpy( beer,"Sierra Nevada Pale Ale");
 vbs[4].set_value( beer);
 vbs[4].get_value( good_beer);
 printf("Good Beer = %s\n",good_beer);
 // get and set an ip an address
 IpAddress ipaddress1, ipaddress2;
 ipaddress1 = "10.4.8.5";
 vbs[5].set_value( ipaddress1);
 vbs[5].get_value( ipaddress2);
 cout << ipaddress2;

}  // end vb example
```

17. Pdu Class

Object Modeling Technique (OMT) view of the SNMP++ Snmp Class



The SNMP++ Pdu class is a C++ encapsulation of the SMI Protocol Data Unit (PDU). PDUs are the basic means of SNMP communication between a manager and an agent. SNMP++ makes working with PDUs easy and safe through use of the Pdu class. The Pdu class allows easy construction, destruction, Vb object loading and unloading of Pdu objects. Since SNMP++ is a bilingual API, the Pdu class is abstract and does not contain SNMP version 1 or version 2 specific information. A single Pdu object can be used for all Snmp class request member functions. The Pdu class is used to interface with the Snmp class for SNMP requests and is also used as a callback parameter for asynchronous requests and notification reception. Note, Pdu objects are zero based in regard to storage of Vbs (the first vb in the Pdu is Vb #0).

For the most part, all Pdu objects are the same in SNMP++. That is, all Pdu objects have identical attributes. The only exception is when using the Pdu objects for sending notifications , traps and informs. For notifications, three additional Pdu Class member functions may be used to set the identity, timestamp and enterprise of a Pdu object.

17.1. Pdu Class Member Functions Overview

Pdu Class Member Functions	Description
Constructors	
Pdu::Pdu(void);	Construct an empty Pdu.
Pdu::Pdu(Vb* pvbs, const int pvb_count);	Construct a Pdu with an array of Vbs and size.
Pdu::Pdu(const Pdu &pdu);	Construct a Pdu with another Pdu.
Member Functions	
int get_vblist(Vb* pvbs, const int pvb_count);	Copies Vbs into caller parameters.
int set_vblist(Vb* pvbs, const int pvb_count);	Sets callers Vbs into the Pdu.
int get_vb(Vb &vb, const int index);	Get a particular Vb out of a Pdu.
int set_vb(Vb &vb, const int index);	Set a particular Vb into a Vb.
int get_vb_count();	Get the Vb count from a Pdu.
int get_error_status();	Get the error status from a Pdu.
int get_error_index();	Get the error index from a Pdu.
unsigned long get_request_id();	Get the request id from a Pdu.
unsigned short get_type();	Get the Pdu type.
int valid();	Is the Pdu valid?
Int delete_vb(const int position);	Removes Vb at specified position from Pdu.
int trim(const int i=1);	Trim off the last Vb in the Pdu,.default 1.
Member Functions For Inform and Trap Usage	
void set_notify_timestamp(const TimeTicks & timestamp);	Set timestamp on a trap or inform Pdu.
void get_notify_timestamp(TimeTicks & timestamp);	Get the timestamp from a trap or inform Pdu object.
void set_notify_id(const Oid id);	Set the ID on a trap or inform Pdu.
void get_notify_id(Oid &id);	Get the ID from a trap or inform Pdu.
void set_notify_enterprise(const Oid &enterprise);	Set the enterprise ID on a trap or inform Pdu.
void get_notify_enterprise(Oid & enterprise);	Get the enterprise ID on a trap or inform Pdu.
Overloaded Operators	
Pdu& operator=(const Pdu &pdu);	Assign one Pdu to another.
Pdu& operator+=(Vb &vb);	Append a Vb to a Pdu;

17.2. Pdu Class Constructors and Destructors

There are a variety of ways to construct Pdu objects including with and without construction parameters.

```
// constructor, no args
Pdu::Pdu( void);

// constructor with Vbs and count
Pdu::Pdu( Vb *vbs, const int vb_count);

// constructor with another Pdu instance
Pdu::Pdu( const Pdu &pdu);

// destructor
Pdu::~Pdu();
```

17.3. Pdu Access Member Functions

The Pdu class supports a variety of member functions for getting and setting Pdu member variables. Included are ways to get and set the variable bindings, error information , request identification, and type information.

```
// extract all the Vbs from a Pdu
int Pdu::get_vblist( Vb* vbs, const int  vb_count);

// deposit Vbs to a Pdu
int Pdu::set_vblist( Vb* vbs, const int vb_count);

// get a particular vb
// where index 0 is the 1st vb
int Pdu::get_vb( Vb &vb, const int index);

// set a particular Vb
// where index 0 is the 1st Vb
int Pdu::set_vb( Vb &vb, const int index);

// return the number of Vbs
int Pdu::get_vb_count();

// return the error index
int Pdu::get_error_index();

// get the error status
int Pdu::get_error_status();

// return the request id
unsigned long Pdu::get_request_id();

// get the Pdu type
unsigned short Pdu::get_type();

// return the validity of a Pdu
int Pdu::valid();
```

17.4. Pdu Class Overloaded Operators

The Pdu class supports overloaded operators for assignment and concatenation of Vb objects to the Pdu.

```
// assignment operator for assigning one Pdu to another
Pdu& operator=( const Pdu &pdu);

// append a Vb object to the Pdu's var bind list
Pdy& operator+=( Vb vb);
```

17.5. Pdu Class Member Functions for Traps and Informs

When working with notifications, traps and informs, SNMP++ provides member functions for getting and setting notification specific values. For usage on these member functions, please refer to the section on sending traps and informs.

```
// set notify timestamp
void Pdu::set_notify_timestamp( const TimeTicks & timestamp);

// get notify timestamp
void Pdu::get_notify_timestamp( TimeTicks & timestamp);

// set the notify id
void Pdu::set_notify_id( const Oid id);

// get the notify id
void Pdu::get_notify_id( Oid &id);

// set the notify enterprise
void Pdu::set_notify_enterprise( const Oid &enterprise);

// get the notify enterprise
void Pdu::get_notify_enterprise( Oid & enterprise);
```

17.6. Loading Pdu Objects

In order to use a Pdu object in a management application, the variable binding list must be loaded into the Pdu instance. This can be done in a variety of manners depending on your needs. Loading a Pdu is typically done before making a SNMP request.

```
// example of how to load a Pdu object
void load_pdu_examples()
{
  Pdu pdu;                    // create a Pdu object
  Vb vb;                      // create a Vb object
  vb.set_oid( SYSDECR);       // set the oid portion of the Vb to System Descriptor
  pdu += vb;                  // loads the Vb to the Pdu

  Pdu my_pdu;                 // create another Pdu object
  Vb vbs[5];                  // create 5 vbs
  pdu.set_vblist( vbs,5);     // load all 5 to the pdu
}
```

17.7. Unloading Pdu Objects

After picking up a response Pdu either from a blocked or asynchronous request, it is necessary to unload the Vbs so that the SMI values may then be extracted.

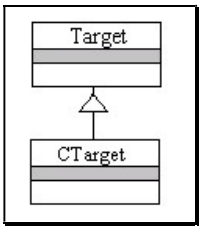
```
// example of how to unload a Pdu
void unload_pdu_example( Pdu &pdu)
{
  ins staus;
  Pdu pdu;                    // create a Pdu object
  Vb vb;                      // create a Vb object
  vb.set_oid(  SYSDECR);      // set the oid portion of the Vb to System Descriptor
  pdu += vb;                  // loads the Vb to the Pdu
  char message[100];          // for the system descriptor printable octet

  Snmp snmp( status);
  if ( status != SNMP_CLASS_SUCCESS) {
    cout < "SNMP++ error = " << snmp.error_msg( status) ;
    return;
  }

  pdu.get_vb( vb,0);          // unload the vb
  vb.get_value( message);     // pull the message out of the vb
  cout << message;            // print it out

};
```

18. SnmpMessage Class

The SnmpMessage class allows Abstract Syntax Notation One (ASN.1) / Basic Encoding Rules (BER) encoding and decoding of SNMP++ objects into SNMP messages ready for transport on the wire. This class allows for easy serialization of Pdu objects which then can be used in any manner desired. Most users of SNMP++ should not have to use this class since the Snmp class takes care of this task including handling timeout and retries. There are cases however where a programmer would like to encode messages. For example, many proxy systems utilize there own transport layer. The SnmpMessage class supports a variety of member functions as shown in the table below.

SnmpMessage Class Member Functions	Description
Constructors	
SnmpMessage::SnmpMessage(void);	Construct an empty SnmpMessage object.
Member Functions	
int load(Pdu pdu, OctetStr community, snmp_version version);	Load a SnmpMessage object with a Pdu, community name and version. Version can be SNMP version 1 or version 2. Returned is the error status.
int load(unsigned char * data, unsigned long len);	Load a SnmpMessage object with a raw data stream. This is useful where a datagram has beeen received across the wire and by loading it into a SnmpMessage object it can then be decoded.
int unload(Pdu &pdu, OctetStr &community, snmp_version &version);	Unload an already loaded SnmpMessage object. This allows getting a Pdu, community and version.
unsigned char * data();	Access the raw ASN.1/BER serialized data buffer. The length of this buffer can be accessed via the ::len() memeber function.
unsigned long len()	Return the length of the raw data buffer.
int valid();	Determine if the SnmpMessage object is valid.

19. Target Class



The Target class is a C++ class used with SNMP++ to allow definition and usage of targets. A target can be thought of as a management definition of an agent to be used for SNMP communication . A target consists of more than just an network address. Targets contain retransmission and time-out policy information, the SNMP protocol type (SNMP version 1 or SNMP version 2) and more. Currently, the only Target sub-class is the Community-based CTarget. The CTarget class may be used for SNMP version 1 and SNMP version 2c communication. This allows your code to be reused for V2 communication with no changes. The Target class allows an SNMP++ session to be independent of a particular agent's attributes.

19.1. Abstract Targets

SNMP++ supports the notion of an abstract Target. This is a Target which may take on any actual derived Target class. All SNMP++ member functions using Targets accept abstract Targets and not specific derived Target objects. This abstract interface allows minimal code change when supporting new Targets.

19.2. Target Addresses

Each target has associated with it a Address object. This Address is a GenAddress and therefor may take on the value of any SNMP++ Address, (IP, IPX or whatever). To specify the address of a managed agent, the agent's address is simply specified and then attached to a Target via construction parameters or through member functions.

19.3. Retransmission Policies

Each Target has a re-transmission policy where a defined time-out and retry specify how long to wait for a SNMP response and how many times to retry if a SNMP response was not heard. Time-out is defined in hundredths of a second where a value of 100 means wait for one second for each response. Retries denote the number of times to retry where the first request is not a retry, it is just a try. So a retry value of 3 equates to retry a maximum of three times when waiting for a response. The total amount of time to wait can be computed as follows.

Total Wait Time = time-out * (retry+1)

If a SNMP++ response does not arrive in the computed total wait time, a SNMP++ time-out error code will be returned. This behavior applies to both blocked and asynchronous calls.

19.4. Target Class Interface

Target Class Member Functions	Description
Constructors	
CTarget::CTarget(void);	Construct an invalid CTarget. Defaults to "public" for community names and retry=1 time_out=1 for re-transmission policy.
CTarget::CTarget(const Address &address , const char *read_community_name, const char *write_community_name,	Construct a CTarget using community names and an Address object. Defaults to retry=1 and time-out =100ms for re-transmission policy.
CTarget(const Address &address const OctetStr &read_community_name, const OctetStr &write_community_name);	Construct a CTarget using OctetStr Communities and an Address.
CTarget::CTarget(Address &address);	Construct a CTarget using an Address object Defaults to "public" for community names and retry=1 time_out=1 for re-transmission policy.
CTarget::CTarget(const CTarget &target);	Copy constructor.
Destructor	
CTarget::~CTarget();	Deletes CTarget object, frees up all resources.
Member Functions	
char * get_readcommunity();	Returns read community name.
void get_readcommunity(OctetStr& read_community_oct);	Gets the read community as OctetStr.
void set_readcommunity(const char * get_community);	Sets the read community.
void set_readcommunity(const OctetStr& read_community);	Set the read community name using an OctetStr.
char * get_writecommunity();	Get the write community.
void get_writecommunity(OctetStr &write_community_oct);	Get the write community as an OctetStr.
void set_writecommunity(const char * new_set_community);	Set the write community.
void set_writecommunity(const OctetStr& write_community);	Set the write community using an OctetStr.
int get_address(GenAddress &address);	Get the Address object.
void set_address(Address &address);	Set the Address portion.
CTarget& operator=(const CTarget& target);	Assign one CTarget to another.
snmp_version get_version();	Return the SNMP version. (version1 or version2c)
void set_version(const snmp_version v);	Set the version. (version1 or version2c)
int operator==(const CTarget &lhs, const CTarget &rhs);	Compare two CTargets.
Abstract Class Member Functions	
int valid();	Returns validity of a Target.
void set_retry(const int r);	Sets the retry.
int get_retry();	Get the retry value.
void set_timeout(const unsigned long t);	Set the time-out value.
unsigned long get_timeout();	Get the time-out value.

19.5. CTarget Class (Community Based Targets)

The CTarget class allows explicit definition of Community based targets. A CTarget defines a SNMP agent using SNMP community based attributes . This includes the *read* and *write community names* and an *address.* The address is represented using the SNMP++ Address class, so an address may be an IP or IPX address. The CTarget class should be used when the application programmer explicitly knows that the agent supports SNMP community based access (SNMP version 1 or SNMP version 2c).

19.5.1. Constructing CTargets

CTarget objects may be instantiated in three different ways.

```
// ----------[ instantiating CTarget Objects ]-----------------------------

// valid complete instantiation
CTarget    ct((IpAddress)"10.10.10.10",  // Address
                 "public",                 // read community name
                 "public");                // write community name

// valid complete using "public" defaults
CTarget ct( (IpAddress) "1.2.3.4");

// invalid CTarget
CTarget ct;
```

19.5.2. Modifying CTargets

```
//----[ modifying CTargets ]-----------------------------------
ct.set_readcommunity("private");   // modifying the read community

ct.set_writecommunity("private");  // modifying the write community

ct.set_address( (IpAddress) "15.29.33.210");
```

19.5.3. Accessing CTargets

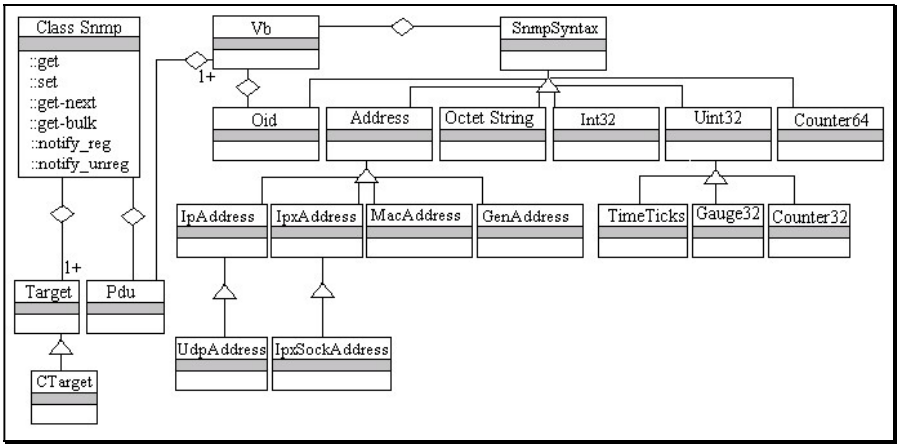
```
//-----[ Accessing CTarget member variables ]-------------------------

// get the write community name
cout << "Write community" << ct.get_writecommunity();

// get the read community name
cout << "Read community " << ct.get_readcommunity();

// get the address
GenAddress address;
ct.get_address( address);

// check the validity of a target
if ( ct.valid())
   cout << "Target is valid";
```

19.5.4. CTarget Examples

```
//------------[CTarget class examples ]-------------------------------------------------------------

// create a valid CTarget using a GenAddress
CTarget ct( (GenAddress) "10.20.30.40");

// create a valid CTarget using an IpxAddress
IpxAddress ipxaddress("01010101-010101010101");
CTarget my_target( ipxaddress);                    // use default "public" for communities

// create an invalid CTarget object
CTarget  ct;                                       // no construction params therefor invalid
if ( !ct.valid())
   cout << "Invalid CTarget instance!";

// get the read community
cout << "Read Community =" << ct.get_readcommunity();

// get the write community
cout << "Write Community = " << ct.get_writecommunity();

// modify the get community
ct.set_readcommunity( "pilsner");

// modify the write community
ct.set_writecommunity("pale ale");
```

20. Snmp Class

Object Modeling Technique (OMT) view of the SNMP++ Snmp Class



The most important class in SNMP++ is the Snmp class. The Snmp class is an encapsulation of a SNMP session. SNMP++ provides a logical binding of a management application with a SNMP++ session to and from specified agents. Handled by the session is the construction, delivery and reception of PDUs. Most APIs require the programmer to directly manage the session. This includes providing a reliable transport mechanism handling time-outs, retries and packet duplication. The SNMP class manages a large part of the session and frees the implementor to concentrate on the agent management. By going through the Snmp class for session management, the implementor is driving through well developed and tested code. The alternative is to design, implement and test your own SNMP engine. The Snmp class manages a session by 1) managing the transport layer over a UDP or IPX connection. 2) handles packaging and un-packaging of variable bindings into PDUs 3) provides for the delivery and reception of PDUs and 4) manages all necessary SNMP resources.

The Snmp class is easy to use. Six basic methods, Snmp::get, Snmp::set, Snmp::get_next, Snmp::get_bulk, Snmp::inform() and Snmp::trap() provide the basic functions for a network management application. Blocking or non-blocking (asynchronous) methods may be used. Multiple sessions may be used each asynchronously firing simultaneous requests. Notification Sending and receiving are supported through the Snmp::trap() and Snmp::inform(), trap and inform sending, and Snmp::notify_register() and Snmp::notify_unregister() for trap and inform reception.

The Snmp class is safe to use. The constructor and destructors allocate and de-allocate all resources needed. This minimizes the likelihood of corrupt or leaked memory. All of the internal SNMP mechanisms are hidden and thus cannot be inadvertently modified.

The Snmp class is portable. The Snmp class interface is portable across OS's and NOS's. The majority of the SNMP++ classes can be compiled and used on any ANSI / ISO C++ compiler. The amount of coding needed to port SNMP++ to another platform is minimal.

20.1. Snmp Class Member Functions Overview

Snmp Class Member Functions	Description
Constructor	
Snmp::Snmp(int &status);	Construct a Snmp object, status defines success.
Destructor	
Snmp::~Snmp( );	Destructor, frees all resources, closes session.
Member Functions	
char * error_msg(const int status);	Returns text string for a given error status.
int get(Pdu &pdu, SnmpTarget &target);	Invokes blocked SNMP get. Gets Pdu from target.
int set(Pdu &pdu, SnmpTarget &target);	Invokes blocked SNMP set. Sets Pdu from target.
int get_next(Pdu &pdu, SnmpTarget &target);	Invokes blocked SNMP get next, using Pdu from target.
int get_bulk(Pdu &pdu, SnmpTarget &target, const int non_repeaters, const int max_reps);	Invokes blocked SNMP get bulk (V2 targets only otherwise uses get-next)
int inform(Pdu &pdu, SnmpTarget &target);	Invoke a blocked mode inform.
int get(Pdu &pdu, SnmpTarget &target, snmp_callback callback, void * callback_data=0);	Invokes SNMP asynchronous get. Gets Pdu from target, uses defined callback and callback data.
int set(Pdu &pdu, SnmpTarget &target, snmp_callback callback, void * callback_data=0);	Invokes SNMP asynchronous set. Sets Pdu from target, uses defined callback and callback data.
int get_next(Pdu &pdu, SnmpTarget &target, snmp_callback callback, void * callback_data=0);	Invokes SNMP asynchronous get next. Get next Pdu from target, uses defined callback and callback data.
int get_bulk(Pdu &pdu, SnmpTarget &target, const int non_repeaters, const int max_reps snmp_callback callback, void * callback_data=0);	Invokes SNMP asynchronous get bulk. Get bulk Pdu from target, uses defined callback and callback data. (V2 targets only otherwise uses get-next)
int inform(Pdu &pdu, SnmpTarget &target, snmp_callback callback, void * callback_data=0);	Invokes asynchronous inform. Uses notify callback.
int trap(Pdu &pdu, SnmpTarget &target);	Sends a trap to the specified target.
int notify_register(TargetCollection &targets, OidCollection &trapids, snmp_callback callback, void * callback_data=0);	Register to receive traps and/or informs.
int notify_register(TargetCollection &targets, OidCollection &trapids, AddressCollection &listen_addresses, snmp_callback callback, void * callback_data=0);	Register to receive traps and/or informs and specify listen interfaces using the AddressCollection.
int notify_unregister();	Un-register to receive traps and/or informs.
int cancel(const unsigned long rid);	Cancels a pending asynchronous request for a given request id.

20.2. Bilingual API

All Snmp class member functions are bilingual. That is they may be used with identical parameter lists for SNMP version 1 or version 2c Targets. This frees the programmer from having to modify code to talk to a SNMP version 2 agent.

20.3. Snmp Class Public Member Functions

The Snmp class provides a variety of member functions for creating, managing and terminating a session. Multiple Snmp objects may be instantiated at the same time.

20.3.1. Snmp Class Constructors and Destructors

The constructors and destructors for the Snmp class allow sessions to be opened and closed. By constructing a Snmp object, a Snmp session is open. UDP or IPX sockets are created and managed until the objects are destroyed. Snmp objects may be instantiated dynamically or statically.

20.3.2. Snmp Class Constructor

A required construction parameter is the return status. Since constructors do not return values in C++, the caller must provide a status which should be checked after instantiating the object. The caller should check the return status for 'SNMP_CLASS_SUCCESS'. If the construction status does not indicate success, the session should not be used.

```
// constructor, blocked SNMP object
Snmp::Snmp(  int &status);              // construction status
```

20.3.3. Snmp Class Destructor

The SNMP class destructor closes the session and releases all resources and memory.

```
// destructor
Snmp::~Snmp();
```

20.3.4. Snmp Class Request Member Functions

In order to access or modify an agent's MIB, requests must be made via the Snmp::get(), Snmp::set(), Snmp::get_next(), Snmp::get_bulk(), Smnp::inform() andSnmp::trap(). All of these member functions accept the similar parameter lists.

20.3.5. Snmp Class Blocked Get Member Function

The get member function allows getting objects from the agent at the specified target. The caller must specify the destination target and requested Pdu.

```
//--------[ get ]-------------------------------------------
int  Snmp::get(  Pdu &pdu,                  // Pdu to get
                 SnmpTarget &target);       // specified target
```

20.3.6. Snmp Class Blocked Get Next Member Function

The get next member function may be used to traverse an agents MIB.

```
//---------[ get next ]--------------------------------------
 int Snmp::get_next( Pdu &pdu,                  // Pdu to get next
                     SnmpTarget &target);       // specified target
```

20.3.7. Snmp Class Blocked Set Member Function

The set member function allows setting agent's objects.

```
//---------[ set ]-------------------------------------------
 int   Snmp::set( Pdu &pdu,                             // Pdu to set
                  SnmpTarget &target);                  // specified target
```

20.3.8. Snmp Class Blocked Get Bulk Member Function

SNMP++ provides a get bulk interface for both SNMP version 1 and version 2 Targets. For SNMP version 1operation, this member function will map over get next.

```
//--------[ get bulk ]-------------------------------------------
int Snmp::get_bulk(  Pdu &pdu,                  // pdu to get_bulk
                     Target &target,            // destination target
                     const int non_repeaters,   // non repeaters
                     const int max_reps);       // maximum reps
```

20.3.9. SNMP Class Blocked Inform Member Function

SNMP++ provides a inform interface for sending informs to V2 agents and managers.

```
//-----------[ inform ]-----------------------------------------------------------
int Snmp::inform( Pdu &pdu,                    // pdu to send
                  SnmpTarget &target);         // destination target
```

Specifying the Id of an Inform

Inform IDs are specified in the same manner as trap IDs. Using the Pdu::set_notify_id() member function, the ID of an inform PDU may be specified. Inform identifiers are represented using Oids. To create a Inform ID, simply create an Oid object with the desired inform ID value. And then attach it to a Pdu using the Pdu::set_notify_id() member function. Conversely, the ID of a inform can be obtained using the Pdu::get_notify_id() member function.

Specifying the TimeStamp on a Inform

To specify the timestamp on ainform PDU, the Pdu::set_notify_timestamp() member function can be used. If a Pdu is sent without calling this member function, a timestamp from the SNMP++ engine will be utilized.

20.4. Snmp Class Asynchronous Member Functions

An Snmp instance can support both blocked mode and asynchronous mode requests. Asynchronous requests return thread of control immediately and do not require that the caller wait for the response. In order to make this happen, a callback routine mechanism is utilized. When making asynchronous requests, the caller must specify a callback function and an optional callback data parameter.

20.4.1. SNMP++ Asynchronous Callback Function Type Definition

```
typedef void (*snmp_callback)( int,            // reason
                               Snmp*,          // session handle
                               Pdu &,          // Pdu passed in
                               SnmpTarget &,   // source target
                               void * );       // callback data
```

20.4.1.1. Description of Callback Parameters

Reason(s) , int
The reason parameter is an integer which describes why the callback was called. The callback may have been called for a variety of reasons including the following.

> *SNMP_CLASS_ASYNC_RESPONSE*: An SNMP response has been received. This is a response to a get, set, get-next, get-bulk or inform. The Pdu parameter holds the actual response PDU and the SnmpTarget parameter holds the target which issued the response.
>
> *SNMP_CLASS_TIMEOUT*: A SNMP++ request has timed out based on the time-out and retry information provided in the target instance. The Pdu parameter holds the original request Pdu for reuse. The SnmpTarget parameter holds the original target.
>
> *SNMP_CLASS_SESSION_DESTROYED*. The session has been destroyed. All pending asynchronous requests were not completed.
>
> *SNMP_CLASS_NOTIFICATION:* A notification, trap or inform request, has arrived. The Pdu object holds the actual notify. The notification id, timestamp and enterprise are available through Pdu member functions Pdu::get_notify_id(), Pdu::get_notify_timestamp() and Pdu::get_notifty_enterprise().

Snmp++ Session, Snmp*
This parameter holds the value of the session which made the request. This allows the session to be reused in time-out or get-next conditions.

Response PDU, Pdu&
This parameter holds the response Pdu for responses, notifies and traps. In the event the reason was a failure, the Pdu parameter holds the original request Pdu. Once the Pdu object goes out of scope, its values are no longer attainable.

Target , SnmpTarget&
This parameter holds the source of the Pdu for responses, notifies and traps. If the reason was a failure, the value is the original target used when the request was made.

Callback data ,void *
When the request was placed, an optional argument may be provided allowing callback data. This information is returned in this parameter if it was specified. If it was not specified, this value is null.

20.4.2. Canceling an Asynchronous Request

SNMP++ allows asynchronous requests to be canceled before they have completed. This is useful for situations where your code may need to exit prematurely or when the specified callback is no longer available. Asynchronous requests are canceled automatically when the Snmp object used to issue the requests is destroyed. When this happens, the specified callback will receive a '*SNMP_CLASS_SESSION_DESTROYED'* reason. Alternatively, an individual asynchronous request may be canceled using the Snmp::cancel() member function. This member function silently cancels the asynchronous request specified with the request_id parameter.

```
//-------------[ cancel a request ]----------------------------------
int Snmp::cancel( const unsigned long rid);
```

20.4.3. Snmp Class Asynchronous Get Member Function

The asynchronous get allows getting SNMP objects from the specified agent. The asynchronous get call will return as soon as the request PDU has been sent. It does not wait for the response PDU. The programmer's defined callback will be called when the response PDU has arrived. The implementation of the callback may utilize the response payload in any desired manner.

```
//------------------------[ get async ]-----------------------------------
 int  Snmp::get ( Pdu &pdu,                        // Pdu to get async
                  SnmpTarget &target,              // destination target
                  snmp_callback callback,          // async callback
                  void * callback_data=0);         // callback data
```

20.4.4. Snmp Class Asynchronous Set Member Function

The asynchronous set member function works in the same manner as the get counter part.

```
//------------------------[ set async ]-----------------------------------
 int Snmp::set( Pdu &pdu,                          // Pdu to set async
                SnmpTarget &target,                // destination target
                snmp_callback callback,            // async callback
                void * callback_data=0);           // callback data
```

20.4.5. Snmp Class Asynchronous Get Next Member Function

The asynchronous get-next member function works in the same manner as does async get and async set.

```
//-----------------------[ get next async ]----------------------------
 int Snmp::get_next( Pdu &pdu,                  // Pdu to get_next
                     SnmpTarget &target,        // destination
                     snmp_callback callback,    // async callback
                     void * callback_data=0);   // callback data
```

20.4.6. Snmp Class Asynchronous Get Bulk Member Function

The asynchronous get-bulk member function works in the same manner as does async get and async set.

```
//-----------------------[ get bulk async ]----------------------------
 int Snmp::get_bulk(Pdu &pdu,                   // Pdu to get_bulk async
                    Target &target,             // destination target
                    const int non_repeaters,    // non repeaters
                    const int max_reps,         // max repetitions
                    snmp_callback callback,     // async callback
                    void * callback_data=0);    // callback data
```

20.4.7. Snmp Class Asynchronous Inform Member Function

```
//--------------------[ inform async ]-----------------------------------------
int Snmp::inform( Pdu &pdu,                     // pdu to send
                  SnmpTarget &target,           // destination target
                  snmp_callback callback,       // callback function
                  void * callback_data=0);      // callback data
```

20.5. SNMP++ Notification Methods

The SNMP++ API supports member functions for both sending and receiving traps.

20.5.1. Sending Traps

Sending traps are a useful part of a manager API which allow for event communication to other management stations.

```
//-----------------------[ send a trap ]----------------------------------
int  Snmp::trap( Pdu &pdu,                          // Pdu to send
                 SnmpTarget &target);               // destination target
```

20.5.1.1. Send Trap Member Function Parameter Descriptions

Pdu &pdu
The Pdu to send. This is where the payload of the trap is contained.

SnmpTarget &target
Where to send the trap.

Specifying the Id of a Trap
Trap Ids are specified in the same manner as Inform Ids. Using the Pdu::set_notify_id() member function, the ID of a trap PDU may be specified. Trap identifiers are represented using the SMI SNMP version 2 traps, which are represented using Oids. SNMP++ predefines the following six generic trap Oids. To create a trapid, simply create an Oid object with the desired trap id value. And the attach it to a Pdu using the Pdu::set_notify_id() member function. Conversely, the id of a trap can be obtained using the Pdu::get_notify_id() member function.

SNMP++ Defined Oid Objects for Generic Trap IDs
coldStart ("1.3.6.1.6.3.1.1.5.1")
warmStart ("1.3.6.1.6.3.1.1.5.2")
linkDown ("1.3.6.1.6.3.1.1.5.3")
linkUp ("1.3.6.1.6.3.1.1.5.4")
authenticationFailure ("1.3.6.1.6.3.1.1.5.5")
egpNeighborLoss ("1.3.6.1.6.3.1.1.5.6")

To send an enterprise specific trap, the caller may specify an Oid which is not in the above generic set.

Specifying the TimeStamp on a Trap
To specify the timestamp on a trap PDU, the Pdu::set_notify_timestamp() member function can be used. If a Pdu is sent without calling this member function, a timestamp from the SNMP++ engine will be utilized.

20.5.1.2. Specifying the Trap Enterprise

Not to be confused with enterprise specific traps, the enterprise of any trap represents where the trap originated from in an agents MIB. Typically this is the System Object Identifier of the trap sender, but in theory it may take on any Oid value. In order to accommodate this parameter, SNMP++ allows setting enterprise using the Pdu::set_notify_enterprise() member function. This is an optional call. If used the enterprise provided will be attached to the Pdu object.

20.5.1.3. Specifying Specific Trap Values for SNMP Version 1 Traps

For specification of a SNMP version 1 specific value, the trapid Oid should be constructed as follows. The last subid of the trapid represents the specific value to use. The second to the last subid should be zero. So, to specify a trap specific value, two extra subids need to be appended, a zero and a value or "0.X". This convention is consistent with RFC 1452 which specifies SNMP version 1 to SNMP version 2 trap mappings.

20.5.2. Receiving Notifications

SNMP++ trap and inform reception allows applications to receive traps and informs based on caller specified filtering. Unlike other SNMP operations, informs and traps are unsolicited events which may occur at any time. Informs and traps are therefor asynchronous events. SNMP++ provides member functions which allow for caller specified filtering of informs and traps. Informs and traps can be filtered based on their type, source and destination.

```
//-----------------------[ register to receive traps and informs]-----------------------------------------
// default form listens on all local interfaces using well known port/ socket #'s
int Snmp::notify_register(OidCollection    &ids,                    // types to listen for
                          TargetCollection &targets,                //  targets to listen for
                          snmp_callback    callback,                // callback to use
                          void *callback_data=0);                                  //optional callback data

//-----------------------[ register to receive traps and informs ]-----------------------------------------
// alternate form, AddressCollection allows local listen interface specification
int Snmp::notify_register(OidCollection    &ids,                    // types to listen for
                          TargetCollection &targets,                //  targets to listen for
                          AddressCollection &local_interfaces,      // interfaces to listen on
                          snmp_callback    callback,                // callback to use
                          void *callback_data=0);                   // optional callback data


//-----------------------[ un-register to get traps and informs]-----------------------------------------
int Snmp::notify_unregister();
```

20.5.2.1. Trap and Inform Registration and the Snmp Class

Each Snmp class instance may be registered for its own unique set of traps / informs. That is , a Snmp object can have its own set of filters and

callback to invoke when a trap or inform arrives meeting the filter criteria. The Snmp::notify_register() member function may be invoked multiple times where each new call clears the previous filter settings. Trap / inform reception for a particular session ceases when either the Snmp:notify_unregister() member function is invoked or when the Snmp instance is destroyed.

20.5.2.2. Snmp::notify_register(), Basic Form

The basic or typical form for notification registration includes notification type, and notification source filtering parameters or an OidCollection and a TargetCollection. Using this form of notify_register() will cause notification listening to occur on all local interfaces. Thus, if the local machine is multi-homed (it has multiple network interfaces) all interfaces will be opened up for notify reception using the well known port / socket numbers. For example, say my machine has two network cards, both running the Internet Protocol (IP) and only one using the Internet Exchange Protocol (IPX). Calling the basic form of notify_register() will listen on both IP interfaces using the well known SNMP trap port and on the IPX interface using the well known trap IPX socket number.

20.5.2.3. Snmp::notify_register(), Alternate Form

The alternate or name overloaded form of notify_register() accepts one additional parameter which allows specification of the local interface to listen for informs, or the AddressCollection. The AddressCollection parameter contains a list of Address objects to listen on including IpAddresses, IpxAddresses, UdpAddresses and IpxSockAddresses. The following is a list of interpretations of Addresses in the Collection and how notify_register() will behave.

AddressCollection Element Behavior Definition

Address Class	Value	Description
IpAddress	Any value except 0.0.0.0	Listen on IP interface specified using well known IP port.
IpAddress	0.0.0.0	Listen on all IP interfaces using well known IP port.
UdpAddress	Any value except 0.0.0.0	Listen on IP interface specified using IP port specified.
UdpAddress	0.0.0.0	Listen on al IP interfaces using IP port specified.
IpxAddress	Any value except 00000000:000000000000	Listen on IPX interface specified using well known IPX socket number.
IpxAddress	00000000:000000000000	Listen on all IPX interfaces using well known IPX socket number.
IpxSockAddress	Any value except 00000000:000000000000	Listen on IPX interface specified using socket number specified.
IpxSockAddress	00000000:000000000000	Listen on all IPX interfaces using socket number specified.

20.5.2.4. Filter Behavior of notify_regsiter()

When filtering, filters behave as follows. If an incoming inform or trap matches any item in the OidCollection of ids **AND** the incoming inform or trap

matches any item in the TargetCollection **THEN** the inform / trap will be forwarded to the callers specified callback. Note, if the OidCollection is empty then all informs will pass the id check, if the TargetCollection is empty, then all informs will pass the Target check.

20.5.3. Filtering Using OidCollection, TargetCollection and AddressCollections

SNMP++ provides three ordered collection classes for putting together collections of Oids, Targets and Addresses. All collection classes operate in the identical manner since they are derived from the same C++ template, SnmpCollection. The Generic operations for the collections are as follows.

Target and Oid Collection Class Member Functions	Description
Constructors	
SnmpCollection::SnmpCollection(void);	Construct an empty Collection .
SnmpCollection::SnmpCollection(const T &t);	Construct a Collection using a single element.
Destructors	
SnmpCollection::SnmpCollection();	Destroy the Collection, free up all resources.
Member Functions	
int size();	Return the size of the collection.
SnmpCollection & operator += (T &t);	Append a element to a Collection.
SnmpCollection & operator = (SnmpCollection &collection);	Assign a Collection to another Collection.
T& operator[](int p);	Access an element in a Collection.
int set_element(const T& i, const int p);	Set an existing element in a Collection.
int get_element(T& i, const int p);	Get a element from a Collection.

20.5.3.1. Making and Using Collections as Filters

Making and using a SnmpCollections as a trap / inform reception filters are easy and straightforward. Three of the arguments for notify registrations are collections, TargetCollection, OidCollection and AddressCollection. To build up these filters, first instantiate a collection and then add elements to it using the overloaded '+=' operator.

```
// example of making trap reception filters

// target collection
TargetCollection my_targets;
my_targets += cisco_router;
my_targets += fore_switch;

// Oid collection
OidCollection my_trapids;
my_trapids += coldStart;
my_trapids += warmStart;

// Address collection
AddressCollection my_addresses;
my_addresses += (IpAddress) "10.4.8.5";
my_addresses += (GenAddress) "01020304:010203040506";
```

20.6. SNMP+ Class Error Return Codes

There are a variety of return codes when using SNMP++. The error codes are common across platforms and may aid the application programmer in finding and detecting error conditions.

20.6.1. Snmp Class Error Message Member Function

If an error occurs during usage of the Snmp member function, the Snmp::error_msg( ) member function may be used to retrieve an friendly error string.

```
//-----------------------[ error message]-----------------------------
char * Snmp::error_msg( const int status);    // returns string for provided status
```

21. Operational Modes

SNMP++ is designed to support a variety of operational modes. The modes of operation, allow for Graphical User Interface (GUI) and console mode applications to be constructed. GUI operational mode cooperates with the existing GUI event systems while console mode operational mode allows for utilization of a custom event system or no event system at all.

21.1.1. Microsoft Windows Event System Operation

For MS-Windows usage, SNMP++ cooperates with the MS-Windows messaging system. Blocked mode calls allow other messages to be processed.

21.1.2. Open Systems Foundation (OSF) X11 Motif Operation

The X11 interface is identical to the MS-Windows interface. Both MS-Windows and X11 versions of SNMP++ support blocked and asynchronous modes of usage. One additional function call is required for an X11 application to register its X11 context with SNMP++ in order for SNMP++ to take advantage of X11's event system. This enables XtAppMainLoop() or similar functions to recognize and dispatch all asynchronous SNMP++ events transparently.

- The context parameter passed in is returned from a call to XtAppInitialize().
- The return value is zero if the function is successful in registering with X11.

```
//----------------[ initialize SNMP++ X11 Context ]--------------------------
int SNMPX11Initialize( XtAppContext context);
```

21.1.3. Non GUI Based Application Operation

A third operational mode for SNMP++ is that for constructing text mode console applications. These types of applications can operate using either blocked mode or asynchronous mode calls. For blocked mode calls, no additional SNMP++ function calls are needed. Asynchronous calls require usage of some event system. For console operational asynchronous mode, SNMP++ offers a set of function calls for reading the currently used file descriptors (socket handles) which the caller may then utilize in their own 'select' call. If a SNMP++ file descriptor has a pending event, the caller may invoke a routine to process all pending events.

SNMPGetFdSets
Used to determine which file descriptors will potentially be active. The function fills in read, write and exception masks which then can be passed to 'select'.

```
//-------[ get file descriptor set from SNMP++ ]-----------------------------------
void SNMPGetFdSets( int &maxfds,              // max # of fds represented
                    fd_set &read_fds,         // mask representing read actions
                    fd_set &write_fds,        // mask representing write actions
                    fd_set &exceptfds);       // mask representing exception actions
```

SNMPGetNextTimeout
Used to determine the time the next time-out event will occur. This value can then be used to as the maximum interval that a blocked operation, such as select, should wait before returning control. The time-out calculation is based on the closest time of all user-registered time-outs and SNMP retransmission time-outs.

```
//---------[ Get the next time-out value ]-------------------------------------------------
unsigned long int SNMPGetNextTimeout( );      // returns value in 1/100 of seconds
```

SNMPProcessPendingEvents
Used to process all currently outstanding events. This function may call any callbacks associated with completed time-outs, file descriptors or outstanding SNMP messages. This function does not block, it handles only events outstanding at the time.

```
//------[ process pending events ]------------------------------------------------------
int SNMPProcessPendingEvents();
```

22. Status and Error Codes

SNMP++ provides two levels of errors when performing Snmp class operations. All Snmp class member functions return a status value. One special error value, “SNMP_CLASS_ERR_STATUS_SET” indicate that the Pdu has an internal error which must be retrieved via the Pdu::get_error_status() member function. All SNMP++ error values may be passed into the Snmp::err_msg() member function for a textual description of the error.

SNMP++ General Errors	**Value**	**Description**
SNMP_CLASS_SUCCESS	0	Success Status
SNMP_CLASS_ERROR	-1	General Error
SNMP_CLASS_RESOURCE_UNAVAIL	-2	New or malloc Failed
SNMP_CLASS_INTERNAL_ERROR	-3	Unexpected internal error
SNMP_CLASS_UNSUPPORTED	-4	Unsupported function
Callback Reasons		
SNMP_CLASS_TIMEOUT	-5	Outstanding request timed out
SNMP_CLASS_ASYNC_RESPONSE	-6	Received response
SNMP_CLASS_NOTIFICATION	-7	Received notification (trap/inform)
SNMP_CLASS_SESSION_DESTROYED	-8	Snmp Object destroyed
Snmp Class Errors		
SNMP_CLASS_INVALID	-10	Snmp member function called on invalid instance
SNMP_CLASS_INVALID_PDU	-11	Invalid Pdu passed to mf
SNMP_CLASS_INVALID_TARGET	-12	Invalid target passed to mf
SNMP_CLASS_INVALID_CALLBACK	-13	Invalid callback to mf
SNMP_CLASS_INVALID_REQID	-14	Invalid request id to cancel
SNMP_CLASS_INVALID_NOTIFYID	-15	Missing trap/inform Oid
SNMP_CLASS_INVALID_OPERATION	-16	Snmp operation not allowed for specified target
SNMP_CLASS_INVALID_OID	-17	Invalid Oid passed to mf
SNMP_CLASS_INVALID_ADDRESS	-18	Invalid address passed to mf
SNMP_CLASS_ERR_STATUS_SET	-19	Agent returned response Pdu with error status set
SNMP_CLASS_TL_UNSUPPORTED	-20	Transport not supported
SNMP_CLASS_TL_IN_USE	-21	Transport in use
SNMP_CLASS_TL_FAILED	-22	Transport Failed

23. Error Status Values

When the SNMP++ return value for a member function returns a value of "SNMP_CLASS_ERR_STATUS_SET", an additional error status value can be obtained via the *Pdu::get_error_status()* member function. These values represent the actual SMI PDU error status values found in RFC 1905. These values may be passed into the *Snmp::error_msg()* member function for a friendly description.

Pdu Error Status Macro	Value	Description
SNMP_ERROR_TOO_BIG	1	Pdu Too Big, see error index
SNMP_ERROR_NO_SUCH_NAME	2	No Such Variable Binding name, see returned error index
SNMP_ERROR_BAD_VALUE	3	Bad Variable Binding Value, see returned error index
SNMP_ERROR_READ_ONLY	4	Variable Binding is read only, see returned error index
SNMP_ERROR_GENERAL_VB_ERR	5	General Variable Binding error, see returned error index
SNMP_ERROR_NO_ACCESS	6	Operation Failure, No Access
SNMP_ERROR_WRONG_TYPE	7	Operation Failure, Bad Type
SNMP_ERROR_WRONG_LENGTH	8	Operation Failure, Bad Length
SNMP_ERROR_WRONG_ENCODING	9	Operation Failure, Wrong Encoding
SNMP_ERROR_WRONG_VALUE	10	Operation Failure, Wrong Value
SNMP_ERROR_NO_CREATION	11	Operation Failure, No Creation
SNMP_ERROR_INCONSIST_VAL	12	Operation Failure, Inconsistent Value
SNMP_ERROR_RESOURCE_UNAVAIL	13	Operation Failure, Resource Unavailable
SNMP_ERROR_COMITFAIL	14	Operation Failure, Commit Failure
SNMP_ERROR_UNDO_FAIL	15	Operation Failure, Undo Failure
SNMP_ERROR_AUTH_ERR	16	Operation Failure, Authorization Error
SNMP_ERROR_NOT_WRITEABLE	17	Operation Failure, Not Writable
SNMP_ERROR_INCONSIS_NAME	18	Operation Failure, Inconsistent Name

24. Snmp Class Examples

In addition to the examples in this section, please refer to the files in the table below for complete programs which can be used as useful command line utilities.

Program Name and Description	File Name
SnmpGet, performs SNMP++ get to v1 and v2 agents.	snmpget.cpp
SnmpNext, peforms SNMP++ getNext to v1 and v2 agents.	snmpnext.cpp
SnmpBulk, performs SNMP++ getBulk to v1 and v2 agents.	snmpbulk.cpp
SnmpSet, performs SNMP++ set to v1 and v2 agents.	snmpset.cpp
SnmpTrap, sends v1 or v2 trap to a manager.	snmptrap.cpp
SnmpWalk, walks an agent's MIB using v1 or v2 via GetBulk.	snmpwalk.cpp

24.1. Getting a Single MIB Variable Example

```
#include "snmp_pp.h"
#define SYSDESCR "1.3.6.1.2.1.1.1.0"              // Object ID for System Descriptor
void get_system_descriptor()
{
  int status;                                     // return status
  CTarget ctarget( (IpAddress) "10.4.8.5");       // SNMP++ v1 target
  Vb vb( SYSDESCR);                               // SNMP++ Variable Binding
  Pdu pdu;                                        // SNMP++ PDU

  //-------[ Construct a SNMP++ SNMP Object ]-------------------------------------
  Snmp snmp( status);                             // Create a SNMP++ session
  if ( status != SNMP_CLASS_SUCCESS) {            // check creation status
    cout << snmp.error_msg( status);              //  if fail, print error string
    return;  }

  //-------[ Invoke a SNMP++ Get ]------------------------------------------------
  pdu += vb;                                      //  add the variable binding
  if (  (status = snmp.get(  pdu, ctarget)) != SNMP_CLASS_SUCCESS)
    cout << snmp.error_msg( status);
  else {
    pdu.get_vb( vb,0);                            // extract the variable binding
    cout << "System Descriptor = "<< vb.get_printable_value();  }  // print out

};
```

24.2. Getting Multiple MIB Variables Example

```
#include "snmp_pp.h"
#define SYSDESCR "1.3.6.1.2.1.1.1.0"              // Object ID for system descriptor
#define SYSOBJECTID "1.3.6.1.2.1.1.2.0"           // Object ID for system object ID
#define SYSUPTIME  "1.3.6.1.2.1.1.3.0"            // Object ID for system up time
#define SYSCONTACT "1.3.6.1.2.1.1.4.0"            // Object ID for system contact
#define SYSNAME "1.3.6.1.2.1.1.5.0"               // Object ID for system name
#define SYSLOCATION "1.3.6.1.2.1.1.6.0"           // Object ID for system location
#define SYSSERVICES "1.3.6.1.2.1.1.7.0"           // Object ID for system services
void get_system_group()
{
  int status;                                     // return status
  CTarget ctarget( (IpAddress) "10.4.8.5");       // SNMP++ v1 target
  Vb vb[7];                                       // a vb for each object to get
  Pdu pdu;                                        // SNMP++ PDU

   //-------[ Construct a SNMP++ SNMP Object ]-----------------------------------
  Snmp snmp( status);                             // Create a SNMP++ session
  if ( status != SNMP_CLASS_SUCCESS) {            // check creation status
    cout << snmp.error_msg( status);              //  if fail, print error string
    return;  }

 //-------[ build up the vbs to get]-------------------------------------------------------------
 vb[0].set_oid( SYSDESCR);
 vb[1].set_oid( SYSOBJECTID);
 vb[2].set_oid( SYSUPTIME);
 vb[3].set_oid(SYSCONTACT);
 vb[4].set_oid( SYSNAME);
 vb[5].set_oid( SYSLOCATION);
 vb[6].set_oid( SYSSERVICES);

 //----[ append all the vbs to the pdu ]---------------------------------------------------
 for ( int z=0;z<7;z++)
    pdu += vb[z];

 //-------[ Invoke a SNMP++ Get ]-----------------------------------------------------
  if (  (status = snmp.get(  pdu, ctarget)) != SNMP_CLASS_SUCCESS)
    cout << snmp.error_msg( status);
 else {
    pdu.get_vbs( vb,7);                           // extract the variable bindings
    for ( int w=0;w<7;w++)
       cout <<  vb[w].get_printable_value() << "\n";  }  // print out the value

};
```

24.3. Setting a Single MIB Variable Example

```
#include "snmp_pp.h"
#define SYSLOCATION "1.3.6.1.2.1.1.6.0"          // Object ID for System location
void set_system_location()
{
  int status;                                    // return status
  CTarget ctarget( (IpAddress) "10.4.8.5");      // SNMP++ v1 target
  Vb vb( SYSLOCATION);                           // SNMP++ Variable Binding
  Pdu pdu;                                       // SNMP++ PDU

  //-------[ Construct a SNMP++ SNMP Object ]-------------------------------------
  Snmp snmp( status);                            // Create a SNMP++ session
  if ( status != SNMP_CLASS_SUCCESS) {           // check creation status
  cout << snmp.error_msg( status);               //  if fail, print error string
  return;
}

  //-------[ Invoke a SNMP++ Set ]------------------------------------------------------
  vb.set_value("Upstairs Mezzanine");                    // add location string to vb
  pdu += vb;                                     //  add the variable binding
  status = snmp.set(  pdu, ctarget);
  cout << snmp.error_msg(status);
}
```

24.4. Setting Multiple MIB Variables Example

```
#include "snmp_pp.h"
 #define SYSCONTACT "1.3.6.1.2.1.1.4.0"          // Object ID for system contact
#define SYSNAME "1.3.6.1.2.1.1.5.0"              // Object ID for system name
#define SYSLOCATION "1.3.6.1.2.1.1.6.0"          // Object ID for system location
void multi_set()
{
  int status;                                    // return status
  CTarget ctarget( (IpAddress) "10.4.8.5");      // SNMP++ v1 target
  Vb vb[3];                                      // a vb for each object to get
  Pdu pdu;                                       // SNMP++ PDU

   //-------[ Construct a SNMP++ SNMP Object ]-------------------------------------
  Snmp snmp( status);                            // Create a SNMP++ session
  if ( status != SNMP_CLASS_SUCCESS) {           // check creation status
    cout << snmp.error_msg( status);             //  if fail, print error string
    return;  }

 //-------[ build up the vbs to get]-------------------------------------------------------
-
 vb[0].set_oid( SYSCONTACT);
 vb[0].set_value("Alan Turing");
 vb[1].set_oid( SYSNAME);
 vb[1].set_value(" The Turing Machine");
 vb[2].set_oid( SYSLOCATION );
 vb[2].set_value(" Cambridge, UK");

 //----[ append all the vbs to the pdu ]-----------------------------------------------
 for ( int z=0;z<3;z++)
    pdu += vb[z];

 //-------[ Invoke a SNMP++ Set ]-----------------------------------------------------
 status = snmp.set(  pdu, ctarget);
 cout << snmp.error_msg( status);
}
```

24.5. Walking a MIB using Get-Next Example

```
#include "snmp_pp.h"                                    // include snmp++ header file
void mib_walk()
{
  int status;                                           // return status
  CTarget target( (IpAddress) "10.4.8.5");              // SNMP++ v1 target
  Vb vb;                                                // a SNMP++ vb
  Pdu pdu;                                              // SNMP++ PDU

   //-------[ Construct a SNMP++ SNMP Object ]-----------------------------------
  Snmp snmp( status);                                   // Create a SNMP++ session
  if ( status != SNMP_CLASS_SUCCESS) {                  // check creation status
     cout << snmp.error_msg( status);                   //  if fail, print error string
     return;  }

 //-------[ set up the first vb ]----------------------------------------------------
 vb.set_oid("1");                                       // get next starting seed
 pdu += vb;                                             // add vb to the pdu

 status = SNMP_CLASS_SUCCESS;
 while ( status == SNMP_CLASS_SUCCESS)
 {
     if (  (status = snmp.get_next(  pdu, ctarget)) == SNMP_CLASS_SUCCESS) {
     pdu.get_vb( vb,0);                                 // extract the vb
     cout << "Mib Object = " << vb.get_printable_oid() << "\n";
     cout <<  "Mib Value = " << vb.get_printable_value() << "\n";
     pdu.set_vb( vb,0);                                 // use last vb as the next one
   }
   else
     cout << "SNMP++ Error = " << snmp.error_msg( status);
 }
};
```

24.6. Sending a Trap Example

```
#include "snmp_pp.h"
void send_trap()
{
  int status;                                          // return status
  CTarget target( (IpAddress) "10.4.8.5");             // SNMP++ v1 target
  Pdu pdu;                                             // SNMP++ PDU

  //-------[ Construct a SNMP++ SNMP Object ]-----------------------------------
  Snmp snmp( status);                                  // Create a SNMP++ session
  if ( status != SNMP_CLASS_SUCCESS) {                 // check creation status
    cout << snmp.error_msg( status);                   //  if fail, print error string
    return;  }

  status = snmp.trap( pdu, target,coldStart);
  cout << " Trap Send Status = " << snmp.error_msg( status);
};
```

24.7. Receiving Traps Example

```
#include "snmp_pp.h"
//----------------[ trap callback function definition ]---------------------------------------------------------
void my_trap_callback ( int reason,                    // reason
                        Snmp* session,                 // session handle
                        Pdu & pdu,                     // trap pdu
                        TimeTicks &timestamp,          // timestamp
                        SnmpTarget &target,            // source of the trap
                        void * cbd)                    // optional callback data
{
  Address *address;
  unsigned char get_cummunity[80], set_community[80];
  unsigned long timeout;
  int retry;

  if ( reason == SNMP_CLASS_TRAP) {
    target.resolve_to_C( get_community,                // get community
                         set_community,                // set community
                         &address,                     // address object
                         timeout,                      // timeout
                         retry);                       // retry
    cout << "Trap Received from << address->get_printable() << "Trap Id = " << trapid.get_printable();
  }
  else
    cout << "Trap Receive Error = " << session->error_msg( reason);

};

//---------------[ trap receive register ]-----------------------------------------------------------------
Snmp *snmp;                                            // dynamic Snmp object
void trap_register()
{
  //----------------[ instantiate an Snmp object, delete when no longer receiving traps ]------------
  int status;
  snmp = new Snmp( status);
  if (( snmp == NULL) || ( status != SNMP_CLASS_SUCCESS))
    cout << "Error constructing Snmp Object\n";
  else
  {
    //-------[ set up two empty collections, empty denotes receive all ]-----------------------------
    TargetCollection targets;
    OidCollection trapids;
    //------[ invoke the regsiter ]-------------------------------------------------------------------
    if ( status = snmp->notify_register(  trapids, targets, & my_trap_callback)) !=
SNMP_CLASS_SUCCESS)
      cout << " Snmp Trap Register Error " << snmp->error_msg( status);
  }
};
```

25. References

[Banker, Mellquist]
Banker Kim, Mellquist Peter E., SNMP++, Connexions, The Interoperability Report, Volume 9, No. 3, March 1995.

[Comer]
Comer, Douglas E. , Internetworking with TCP/IP, Principles, Protocols and Architecture, Volume I Prentice Hall, 1991.

[Gama, Helm, Johnson, Vlissides]
Erich Gama, Richard Helm , Ralph Johnson, John Vlissides , Design Patterns, Addison Wesley, 1995.

[Meyers]
Meyers, Scott, Effective C++, Addison Wesley, 1994.

[Petzold]
Petzold Charles, Programming MS-Windows, Microsoft Press

[RFC 1452]
J. Case, K. McCloghrie, M. Rose, S. Waldbusser, Coexistence between version 1 and version 2 of the Internet-standard Network Management Framework, May 03, 1993.

[RFC 1442]
J. Case, K. McCloghrie, M. Rose, S. Waldbusser, Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2), May 03 , 1993.

[Rose]
Rose, Marshall T. , The Simple Book, An Introduction to Internet Management , Second Edition, Prentice Hall Series 1994.

[Rumbaugh]
Rumbaugh, James, Object-Oriented Modeling and Design, Prentice Hall, 1991.

[Saks]
Saks, Dan, C++ Programming Guidelines, Thomas Plum & Dan Sacks, 1992.

[Stallings]
Stallings, William, SNMP, SNMPv2 and CMIP The Practical Guide to Network Management Standards, Addison Wesley, 1993.

[Stroustrup]
Stroustrup , Bjarne, The C++ Programming Language, Edition #2 Addison Wesley, 1991.

[WinSNMP]
WinSNMP, Windows SNMP An Open Interface for Programming Network Management Application under Microsoft Windows. Version 1.1.

[WinSockets]
WinSockets, Windows Sockets, An Open Interface for Network Programming under Microsoft Windows.