Windows SNMP

An Open Interface for Programming Network Management Applications using the Simple Network Management Protocol under Microsoft® Windows

WinSNMP/Manager API

Version 1.1a

9 August 1995

Bob Natale
American Computer & Electronics Corporation

Copyright © 1993, 1994, 1995 by Bob Natale,
American Computer & Electronics Corporation

All rights reserved.

This document may be freely redistributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included. Comments or questions may be submitted via electronic mail to **winsnmp@mailbag.intel.com**. Requests to be added to the Windows SNMP mailing list should be addressed as follows:

To: Majordomo@mailbag.intel.com
Subject: <leave blank>
subscribe WinSNMP

This specification and other information on Windows SNMP are available via anonymous FTP from the host SunSITE.unc.edu under the directory path /pub/micro/pc-stuff/ms-windows/WinSNMP.

Questions about products conforming to this specification should be addressed to the vendors of the products.

Author's Contact Information
Bob Natale
Director, Network Management Products
American Computer & Electronics Corporation
209 Perry Parkway
Gaithersburg MD 20877
301-258-9850 (Tel)
301-921-0434 (Fax)
natale@acec.com (Internet)

Abbreviated Revision History

Rev	Date	Editor	Editor's Address
0.3	January 25, 1993	Microsoft	dwaink@microsoft.com
1.0a	April 28,1993	Amatzia Ben-Artzi	amatzia@netmanage.com
1.0	September 13, 1993	Bob Natale	natale@acec.com (working)
1.1a	DRAFT February 14, 1994	Bob Natale	natale@acec.com (bake-off)
1.1b	DRAFT May 16,1994	Bob Natale	natale@acec.com (draft)
1.1	June 12, 1994	Bob Natale	natale@acec.com (production)
1.1a	August 9, 1995	Bob Natale	natale@acec.com (clarifications)

WinSNMP/Manager API Specification

Table of Contents

1. INTRODUCTION 6
  1.1. What is Windows SNMP? 7
  1.2. Compliance 8
  1.3. SNMP 9
  1.4. Microsoft Windows 10
  1.5. The Status of this Specification 11
  1.6. References 12
    1.6.1. Primary Sources 12
    1.6.2. Secondary Sources 15
  1.7. Glossary: Abbreviations, Acronyms, and Definitions 16
  1.8. Contributors 18
    1.8.1. Special Acknowledgments 18
    1.8.2. Individual Contributors 19
2. PROGRAMMING WITH WINDOWS SNMP 20
  2.1. Levels of SNMP Support 21
    2.1.1. Implementations 21
      2.1.1.1. "Level 0" Implementations 21
      2.1.1.2. "Level 1" Implementations 21
      2.1.1.3. "Level 2" Implementations 22
      2.1.1.4. "Level 3" Implementations 22
    2.1.2. Applications 22
  2.2. Transport Interface Support 23
  2.3. Entity/Context Translation Modes 24
    2.3.1. SNMPAPI_TRANSLATED Mode 24
    2.3.2. SNMPAPI_UNTRANSLATED_V1 Mode 24
    2.3.3. SNMPAPI_UNTRANSLATED_V2 Mode 25
  2.4. Local Database 26
  2.5. Sessions 27
  2.6. Memory Management 28
    2.6.1. HANDLE'd Resources 28
    2.6.2. C-Style Strings 29
    2.6.3. Descriptors 29
    2.6.4. Summary 30
  2.7. Asynchronous Model 31
  2.8. Polling and Retransmission 32
  2.9. RequestIDs 34
  2.10. Error Handling 35
    2.10.1. Common Error Codes 35
    2.10.2. Context-Specific Error Codes 36
    2.10.3. Transport Error Reporting 37
  2.11. WinSNMP Data Types 38
    2.11.1. Integers 39
    2.11.2. Pointers 39
    2.11.3. Function Returns 39

2.11.4. Descriptors 39
3. WINDOWS SNMP INTERFACES 40
3.1. Local Database Functions 41
3.1.1. SnmpGetTranslateMode() 42
3.1.2. SnmpSetTranslateMode() 43
3.1.3. SnmpGetRetransmitMode() 44
3.1.4. SnmpSetRetransmitMode() 45
3.1.5. SnmpGetTimeout() 46
3.1.6. SnmpSetTimeout() 47
3.1.7. SnmpGetRetry() 48
3.1.8. SnmpSetRetry() 49
3.2. Communications Functions 50
3.2.1. SnmpStartup() 51
3.2.2. SnmpCleanup() 53
3.2.3. SnmpOpen() 54
3.2.4. SnmpClose() 55
3.2.5. SnmpSendMsg() 56
3.2.6. SnmpRecvMsg() 58
3.2.7. SnmpRegister() 60
3.3. Entity/Context Functions 62
3.3.1. SnmpStrToEntity() 63
3.3.2. SnmpEntityToStr() 64
3.3.3. SnmpFreeEntity() 65
3.3.4. SnmpStrToContext() 66
3.3.5. SnmpContextToStr() 68
3.3.6. SnmpFreeContext() 70
3.4. PDU Functions 71
3.4.1. SnmpCreatePdu() 73
3.4.2. SnmpGetPduData() 75
3.4.3. SnmpSetPduData() 77
3.4.4. SnmpDuplicatePdu() 78
3.4.5. SnmpFreePdu() 79
3.5. Variable Binding Functions 80
3.5.1. SnmpCreateVbl() 82
3.5.2. SnmpDuplicateVbl() 83
3.5.3. SnmpFreeVbl() 84
3.5.4. SnmpCountVbl() 85
3.5.5. SnmpGetVb() 86
3.5.6. SnmpSetVb() 87
3.5.7. SnmpDeleteVb() 88
3.6. Utility Functions 90
3.6.1. SnmpGetLastError() 91
3.6.2. SnmpStrToOid() 92
3.6.3. SnmpOidToStr() 93
3.6.4. SnmpOidCopy() 94
3.6.5. SnmpOidCompare() 95
3.6.6. SnmpEncodeMsg() 97
3.6.7. SnmpDecodeMsg() 99
3.6.8. SnmpFreeDescriptor() 101
4. Declarations 102
Appendix A. Mapping Traps Between SNMPv1 and SNMPv2 113
Appendix B. Usage Example 116

Appendix C. WinSNMP++ Prototype 117

1. INTRODUCTION

The Windows SNMP API specification defines a programming interface for network management applications running under the Microsoft Windows family of GUI/operating system products, enabling those applications to make use of a logically external SNMP engine or service layer.

For the purpose of exposition, the original Internet-standard Network Management Framework, as described in RFCs 1155, 1157, and 1212, is termed the SNMP version 1 framework (SNMPv1). The new framework that is currently a proposed Internet standard, as described in RFCs 1441, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, and 1452, is termed the SNMP version 2 framework (SNMPv2). In addition, there are three proposed Internet standards, as described in RFCs 1418, 1419, and 1420, that address the use of transports other than UDP over IP for SNMPv1. These RFCs describe SNMPv1 over OSI, AppleTalk, and IPX. Note that these transports are directly addressed in SNMPv2 by RFC 1449.

The Windows SNMP API specification introduces no constraints on the use of SNMPv1 or SNMPv2, nor on the functionality supported by those protocols as prescribed in the relevant Internet RFCs.

For the purposes of this specification, SNMPv1 is seen as a subset of SNMPv2.

Hereinafter the terms "WinSNMP", “WinSNMP/Manager”, and "Windows SNMP" will be used as shorthand for "the Windows SNMP Manager API Specification" (which is the full and formal name for this document itself).

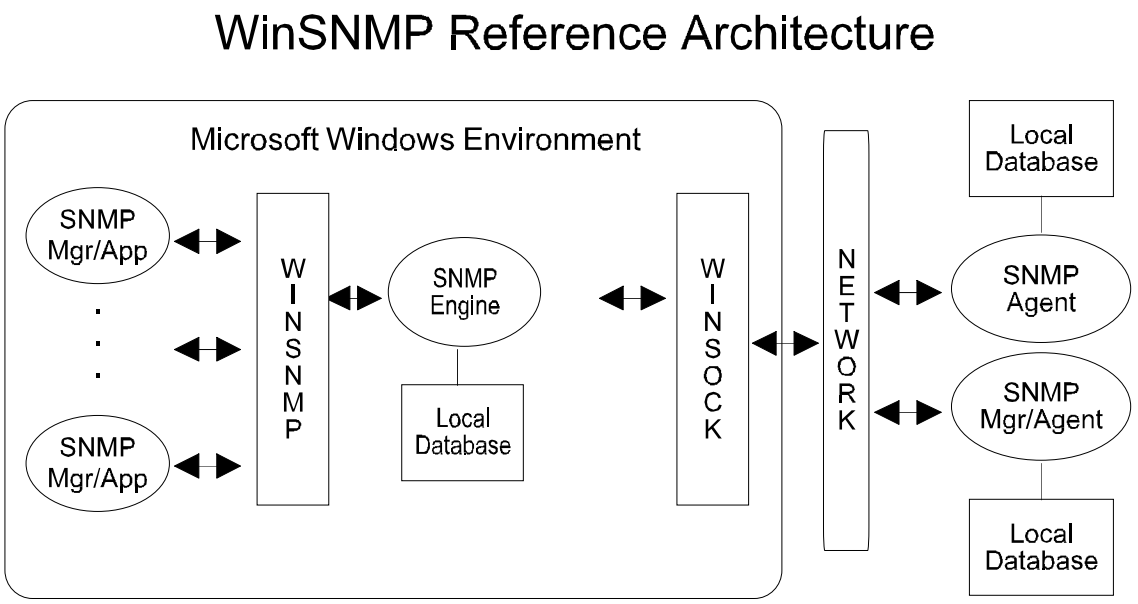
This specification avoids introducing new or different meanings for terms which have established definitions in the existing SNMP literature (especially the RFCs). Readers are encouraged to consult the "Glossary" and "References" sections (and to point out lapses in terminological correctness).

1.1. What is Windows SNMP?

The purpose of WinSNMP is to promote the development of SNMP-based network management applications running under the Microsoft Windows family of GUI/operating system products.

WinSNMP provides a single interface to which application developers can program and multiple SNMP software vendors can conform. This specification thus defines the procedure calls, data types, data structures, and associated semantics to which an application developer can program and which an SNMP software vendor can implement.

The following diagram shows where WinSNMP fits in one possible scenario of end-to-end SNMP connectivity from an entity acting in a manager role (far left) to an entity acting in an agent role (far right). This diagram is a high-level rendition of the model embodied in the current version of WinSNMP. Other models are both possible and supported by the specification, particularly as regards network transport independence.



In summary, WinSNMP offers these major benefits--all intended to accelerate the development, dissemination, and use of SNMP network management applications:

- SNMP enabling technology for functional network management applications (i.e., “hides” ASN.1, BER, and SNMP protocol details).

- SNMP service provider independence. A WinSNMP application will run against any compliant WinSNMP implementation.

- Uniform SNMPv1 and SNMPv2 support. A WinSNMP application does not have to know the SNMP version level of the target SNMP entities acting in an agent role. The WinSNMP implementation will perform any and all necessary mappings between SNMPv1 and SNMPv2 in accordance with the appropriate RFCs.

1.2. Compliance

Software which conforms to this Windows SNMP specification will be considered "WinSNMP compliant".

Suppliers of implementations which are "WinSNMP compliant" shall be referred to as "WinSNMP suppliers". Nothing in the WinSNMP specification is meant to dictate or preclude particular implementation strategies. This specification allows for various overlapping levels of SNMP support on the part of an implementation:

- **Level 0** = Message encoding/decoding only
- **Level 1** = Level 0 + interaction with SNMPv1 agents
- **Level 2** = Level 1 + interaction with SNMPv2 agents
- **Level 3** = Level 2 + interaction with other SNMPv2 managers

To be WinSNMP compliant, a vendor must implement 100% of this WinSNMP specification, as appropriate to the level of SNMP interaction the given implementation supports. WinSNMP vendors are encouraged to state clearly the level of SNMP interaction they support in all of their marketing and technical literature.

Applications which are capable of operating with any "WinSNMP compliant" implementation which supports at least the level of SNMP interaction required by the application will be considered as having a "WinSNMP interface" and will be referred to as "WinSNMP applications".

This version of the WinSNMP specification defines and documents the use of the API by management applications. A future revision or separate extension may include features for use by SNMP agents. A companion document, the “WinSNMP/MIB API Specification”, provides definitions of elements used as operands to SNMP operations.

1.3. SNMP

SNMP is a request-response protocol used to transfer management information between entities acting in a manager role and entities acting in an agent role. Managers are often configured as management stations and agents are often configured as managed nodes. A manager can also act as an agent to another manager in both vertical (hierarchical) and horizontal (distributed) relationships. Likewise, a physical node might be managed by multiple agents, and an agent might manage multiple physical nodes. When hereinafter we use the prototypical management station/managed node perspective for the sake of simplicity and clarity of presentation, that practice is not meant to preclude other forms of SNMP interactions.

Each managed device or application contains monitoring and (possibly) control instrumentation. This instrumentation is accessed by the agent. The agent represents its access to this instrumentation to the manager via a MIB, filtered by the SNMP security mechanisms. Management applications communicate with agents via SNMP to monitor and (possibly) control managed devices or applications.

A management application may issue several requests to an agent, without waiting for a response. Alternatively, it may issue a request and wait for a response, operating in a lock-step fashion with the agent. Furthermore, SNMP may be implemented on a wide range of transport protocols, each with varying delivery mechanisms and reliability characteristics. The normal transmission mechanism (UDP) is through non-guaranteed messages which may be dropped, duplicated, or re-ordered. Thus, with SNMP, it is the responsibility of each management application to determine and implement the desired level of reliability for its communications. This means that the management application decides on its own retransmission and timeout strategy.

Note, also, that an agent may send asynchronous messages--called traps (SNMPv1) or notifications (SNMPv2)--to a management application. This important feature of SNMP is also fully supported by WinSNMP. Note that in this document, the term "traps" is used to refer both to traps and notifications, unless specifically qualified as otherwise in a given instance.

1.4. Microsoft Windows

This API is intended to be usable within all implementations and versions of Microsoft Windows "family" of operating systems and graphical user interface environments from Microsoft Windows Version 3.0 onwards, including Windows v3.1, Windows for Workgroups v3.11, and Windows NT (among others). It thus provides for WinSNMP implementations and WinSNMP applications in both 16- and 32-bit operating environments, and in both single- and multi-threaded execution environments.

WinSNMP makes provisions for multi-threaded Windows processes. A process contains one or more threads of execution. In the Win16 non-multi-threaded world, a task corresponds to a process with a single thread. All references to threads in this document refer to actual "threads" in multi-threaded Windows environments. In single-threaded environments (such as Windows 3.0 and Windows 3.1), use of the term thread refers to a Windows process.

1.5. The Status of this Specification

This specification is released for stable implementation as of this revision (v1.1). It is the product of collaboration among an informal, *ad hoc*, self-organized group of vendors, developers, and users with an interest in SNMP-based network management applications and the Microsoft Windows family of development and runtime supporting environments for such applications.

Readers of this specification are assumed to have a high degree of knowledge and understanding of SNMP (through SNMPv2) and Microsoft Windows programming conventions. Contributions aimed at reducing the level of detailed knowledge of these specific technologies required for users of this specification are invited.

At the present time, it is believed that this version (v1.1) of the WinSNMP/Manager API specification permits the development of interoperable implementations and applications supporting at least "SNMP Level 1" as defined herein and that applications developed in accordance with this version of the specification will, additionally, be structured for future compatibility through "SNMP Level 3" as defined herein. Due to limitations in the understanding and specification of "local database" functions in the current version, it is unlikely that full SNMPv2 operational support can be achieved without additional functions and data structures being defined. It is expected that implementation, development, and interoperability experience gained with this v1.1 of this specification will identify those additions, and that they will appear in v2.0 of the specification in early 1995.

Consensus on the release of the previous version of the specification was reached on the mailing list and confirmed at the third BOF meeting of the WinSNMP group held at the August '93 Interop in San Francisco. This was followed by an interoperability testing event, hosted by NetManage in Cupertino in February 1994. On-going edits to v1.1 will be accepted to:

1. correct errors and inconsistencies;

2. add explanatory and/or expository text and graphics;

3. add essential new functions, variables, error codes, data types, or data structures required for usability of the baseline specification.

Contributions oriented toward new and/or expanded functionality outside the scope of the preceding three objectives will be accepted, but will be considered for inclusion in v2.0 of this specification.

1.6. References

This section consists of two parts: Primary sources (mainly RFCs) and secondary sources.

1.6.1. Primary Sources

RFC	Title	Author(s)	Date	Comments
1089	SNMP over Ethernet	M.L. Schoffstall, C. Davin, M. Fedor, J.D. Case	Feb 1989	none
1098	Simple Network Management Protocol (SNMP)	J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin	Apr 1989	OBSOLETES: RFC1067, OBSOLETED-BY: RFC1157
1155	Structure and Identification of Management Information for TCP/IP-based Internets	M. T. Rose, K. Z. McCloghrie	May 1990	OBSOLETES: RFC1065
1156	Management Information Base for Network Management of TCP/IP-based Internets	K. Z. McCloghrie, M. T. Rose	May 1990	OBSOLETES: RFC1066
1157	Simple Network Management Protocol (SNMP)	J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin	May 1990	OBSOLETES: RFC1098
1158	Management Information Base for Network Management of TCP/IP-based Internets: MIB-II	M. T. Rose	May 1990	OBSOLETED-BY: RFC1213
1161	SNMP over OSI	M.T. Rose	Jun 1990	OBSOLETED-BY: RFC1418
1187	Bulk table retrieval with the SNMP	M.T. Rose, K. McCloghrie, J.R. Davin	Oct 1990	none
1213	Management Information Base for Network Management of TCP/IP-based Internets: MIB-II	K. Z. McCloghrie, M. T. Rose	Mar 1991	OBSOLETES: RFC1158
1215	Convention for defining traps for use with the SNMP	M.T. Rose	Mar 1991	none
1227	SNMP MUX protocol and MIB	M.T. Rose	May 1991	none
1270	SNMP communications services	F. Kastenholz	Oct 1991	None
1283	SNMP over OSI	M. Rose	Dec 1991	OBSOLETED-BY: RFC1418
1284	Definitions of Managed Objects for the Ethernet-like Interface Types	J. Cook	Dec 1991	none
1285	FDDI Management Information Base	J. Case	Jan 1992	none

1286	Definitions of Managed Objects for Bridges	E. Decker, P. Langille, A. Rijsinghani, K. McCloghrie	Dec 1991	none
1289	DECnet Phase IV MIB Extensions	J. Saperia	Dec 1991	none
1298	SNMP over IPX	R. Wormley, S. Bostock	Feb 1992	OBSOLETED-BY: RFC1420
1303	A Convention for Describing SNMP-based Agents	K. McCloghrie, M. Rose	Feb 1992	SEE-ALSO: RFC1155, RFC1212, RFC1213, RFC1157
1351	SNMP Administrative Model	J. Davin,J. Galvin,K. McCloghrie	Jul 1992	none
1352	SNMP Security Protocols	J. Galvin,K. McCloghrie,J. Davin	Jul 1992	none
1381	SNMP MIB Extension for X.25 LAPB	D. Throop, F. Baker	Nov 1992	none
1382	SNMP MIB Extension for the X.25 Packet Layer	D. Throop	Nov 1992	none
1407	Definitions of Managed Objects for the DS3/E3 Interface Type	Tracy A. Cox, Kaj Tesink	Jan 1993	OBSOLETES: RFC1233
1414	Identification MIB	M. StJohns & M. Rose	Jan 1993	none
1418	SNMP over OSI	M. Rose	Feb 1993	OBSOLETES: RFC1161, RFC1283
1419	SNMP over AppleTalk	G. Minshall & M. Ritter	Feb 1993	none
1420	SNMP over IPX	S. Bostock	Feb 1993	OBSOLETES: RFC1298
1441	Introduction to version 2 of the Internet-standard Network Management Framework	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none
1442	Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2)	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none
1443	Textual Conventions for version 2 of the Simple Network Management Protocol (SNMPv2)	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none
1444	Conformance Statements for version 2 of the Simple Network Management Protocol (SNMPv2)	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none
1445	Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2)	J. Galvin & K. McCloghrie	Apr 1993	none

1446	Security Protocols for version 2 of the Simple Network Management Protocol (SNMPv2)	J. Galvin & K. McCloghrie	Apr 1993	none
1447	Party MIB for version 2 of the Simple Network Management Protocol (SNMPv2)	K. McCloghrie & J. Galvin	Apr 1993	none
1448	Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2)	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none
1449	Transport Mappings for version 2 of the Simple Network Management Protocol (SNMPv2)	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none
1450	Management Information Base for version 2 of the Simple Network Management Protocol (SNMPv2)	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none
1451	Manager-to-Manager Management Information Base	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none
1452	Coexistence between version 1 and version 2 of the Internet-standard Network Management Framework	J. Case, K. McCloghrie, M. Rose, & S. Waldbusser	Apr 1993	none

1.6.2. Secondary Sources

Black, Uyless D. *TCP/IP and Related Protocols*. McGraw-Hill, 1992.

Comer, Douglas E., and Stevens, David L. *Internetworking with TCP/IP - Volume II: Design, Implementation, and Internals.* Prentice-Hall, 1991. (Chaps. 18-20.)

Feit, Sidnie. *SNMP: A Guide to Network Management*. McGraw-Hill, 1994.

Feit, Sidnie. *TCP/IP: Architecture, Protocols, and Implementation.* McGraw-Hill, 1993.

Harnedy, Sean J. *Total SNMP: Exploring the Simple Network Management Protocol.* CBM Books, 1994.

Leinwand, Allan, and Fang, Karen. *Network Management: A Practical Perspective*. Addison-Wesley, 1993.

Miller, Mark E., P.E., *Managing Internetworks with SNMP: The Definitive Guide to the Simple Network Management Protocol (SNMP) and SNMP version 2*. M&T Books, 1993.

Perkins, David T. "Understanding SNMP MIBS". Rev. 1.1.5, July 7, 1992.6, September, 1993.

Rose, Marshall T.. and McCloghrie, Keith Z. *How to Manage Your Network Using SNMP: The Network Management Practicum*. Prentice-Hall, 1995 (published in 1994).

Rose, Marshall T. *The Simple Book: An Introduction to Management of TCP/IP-based Networks.* Prentice-Hall, 1990.

Rose, Marshall T. *The Simple Book: An Introduction to Internet Management.* 2nd ed. Prentice-Hall, 1994 (published in 1993).

Stallings, William. *SNMP, SNMPv2, and CMIP: The Practical Guide to Network Management Standards.* Addison-Wesley, 1993.

Steedman, Douglas. *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference*. Technology Appraisals, Ltd., 1990 (with *Errata* sheet dated March 14, 1991). ISBN 1-871802-06-7.

1.7. Glossary: Abbreviations, Acronyms, and Definitions

The following table of abbreviations and definitions presents terms frequently used in the SNMP RFCs and related literature along with the official, customary, or consensus meaning(s). An editing objective of this specification is to use none of these terms in any sense other than that shown in the corresponding definition, nor to invent or employ new terms with meanings functionally equivalent to those of these established terms.

Short	Term	Definition
	Access Control	Restriction of access to MIB objects either in scope or function or both, on the basis of party.
ACL	Access Control List	An indication of what actions (aclPrivileges) may be performed by a given party (aclTarget) on behalf of another party (aclSubject) within a given context (aclResources).
API	Application Programming Interface	A defined set of procedure calls, data types, data structures, and associated semantics used to incorporate a logically external service layer into an application program.
	Authentication	Verification of message source and timeliness. Achieved in SNMPv2 normally by message component transformation via secret codes.
	Authorization	Defining and controlling the levels of legitimate access to data and/or resources. Achieved in SNMPv2 with the use of ACLs, and in SNMPv1 with the use of Community strings.
	Bilingual Entity	A protocol entity that can handle both SNMPv1 and SNMPv2 operations, semantics, and communications.
	Community	An administrative relationship between SNMPv1 entities; identified by a "community string".
CTX	Context	A collection of managed object resources accessible by an SNMP entity.
	Dual-Role Entity	A protocol entity capable of acting in both a manager and an agent role.
Entity	Protocol Entity	An SNMP-conversant process, operating in either and agent role or in a manager role, or both, which performs network management operations by generating and/or responding to SNMP protocol messages.
GUI	Graphical User Interface	A visually rich user interface (as contrasted with a Command Line Interface [CLI]).
	Local Database	An entity-specific collection of information about parties, contexts, views, and access control policies.
M2M	Manager-to-Manager MIB	Used to define conditions and thresholds at a manager that will trigger notifications to other managers.

MIB	Management Information Base	A virtual information store holding a collection of managed object definitions.
View	MIB View	A subset of the universal set of all instances of all MIB objects; defined as a collection of view subtrees.
	Mid-Level Manager	A dual-role protocol entity acting primarily in a manager role except when acting in an agent role vis-à-vis other managers.
OID	Object Identifier	A data type denoting an authoritatively named object; a sequence of non-negative integers.
	Party	A conceptual, virtual execution environment who operation is restricted to an administratively defined subset of all possible operations of a particular entity. A set of specific security characteristics.
	Party MIB	Used to configure parties at protocol entities with various security characteristics, including use or non-use of authentication and/or privacy and access control constraints.
	Privacy	Protection of transmitted data from eavesdropping. Achieved in SNMPv2 normally by message component encryption.
PDU	Protocol Data Unit	A data structure containing SNMP message components and used for communication between SNMP entities.
Proxy	Proxy Agent	
RFC	Request for Comments	The document series describing the Internet suite of protocols and related experiments.
SNMP	Simple Network Management Protocol	The application protocol offering network management service in the Internet suite of protocols. Abbreviation used for both SNMPv1 and SNMPv2.
variable	SNMP variable	An object's identity and its instance value encoded as an OID.
varbind	Variable-Binding	A pairing of an object instance name and an associated value or pseudo-value and syntax.
varbindlist	Variable-Bindings List	A grouping of one or more variable-bindings

1.8. Contributors

1.8.1. Special Acknowledgments

Special acknowledgment is made to the following individuals and organizations for critical contributions to the origination, evolution, and realization of the WinSNMP/Manager API:

- Amatzia Ben-Artzi, NetManage, for contributing original SNMP API specifications to be used a starting point for the WinSNMP/Manager API and for hosting the first WinSNMP interoperability tests in February 1993.

- Dwain Kinghorn, Microsoft, for contributing original SNMP API specifications to be used as a starting point for the WinSNMP/Manager API.

- Marshall Rose, Dover Beach Consulting, for a major enhancement to the early WinSNMP/ Manager API to provide for transparent SNMPv1 and SNMPv2 support and to enable transport independence. Marshall, of course, is a beacon of SNMP understanding and a bastion of SNMP authority.

- Two other notable SNMP authorities--Jeff Case of SNMP Research and Dave Perkins of Synoptics--contributed extensive reviews of the pre-1.0 versions of this specification. In addition, three other equally qualified SNMP experts--Bob Stewart of Xyplex, Keith McCloghrie of Cisco, and Karl Auerbach--provided invaluable insights and guidance at various stages of the working groups deliberations.

- Bob Natale, American Computer & Electronics Corp, for serving as group moderator and editor of the WinSNMP family of APIs--WinSNMP/Manager, WinSNMP/MIB, and WinSNMP/Agent.

- Mark Towfiq and Simon Spero, SunSITE, for their generosity, patience, and effort with respect to WinSNMP mailing list and file archive administration.

Also, several specific technical contributions deserve mention: Maria Greene (while at Xyplex) for the “strawman” C++ Appendix; Mary Quinn of FTP Software for the transport layer error codes; Reuben Sivan of MultiPort Software for the “const” types; and Kee Lai of FTP Software for the trap processing example Appendix. Each of those contributions involved not just an idea expressed via e-mail, but some actual work in writing and testing proposed code. There have been many good ideas expressed on the WinSNMP mailing list by many people--hopefully all of them are listed below. However, one additional individual deserves special mention here because he participated in almost all of the technical discussions with patience, imagination, and persistence...many of the low-level specifics of the WinSNMP/Manager API have been affected (positively :-) by the input of Chris Young of Cabletron.

In addition, the individuals listed in the next section and their sponsoring organizations have contributed ideas, time, and (in some cases) other valuable resources, to the creation of the Windows SNMP API specification. (Other individuals have indicated a desire to help with some of the planned extensions to the WinSNMP/Manager API...look for their names in the v2.0 document next year!)

1.8.2. Individual Contributors

Tom Abraham	abraham@acec.com
Ed Alcoff	oldera@nat.com
James Allard	jallard@microsoft.com
Karl Auerbach	auerbach@ssds.com
Larry Backman	backman@ftp.com
John Bartas	jbartas@sunlight.com jrb@ibeam.ht.intel.com
Amatzia Ben-Artzi	amatzia@netmanage.com
Chris Bologna	chris@distinct.com
Jeff Case	case@snmp.com
Seung “Tae” Chin	schin@novell.com
David Corbello	corb@acc.com
William Dunn	willliy@netmanage.com
Hope Fabian	hope@ralvm12.vnet.ibm.com
Nick Gandin	gandin@acec.com
Maria Greene	maria@maelstrom.timeplex.com
Michael Greenberg	arnoff@ftp.com
Jim Greuel	j_greuel@hpcnd.cnd.hp.com
Dwain Kinghorn	dwaink@microsoft.com
Guenther Kroenert	Guenther.Kroenert@sto.mchp.sni.de
Kee Lai	klai@ftp.com
Ray C. Langford	ray@frontiertech.com
Osip Liitsci	osip@netmanage.com
Keith McCloghrie	kzm@cisco.com
Evan McGinnis	bem@nsd.3com.com
Victoria J. McGuire	victoria@ralvm12.vnet.ibm.com
John F. Moehrke	john@frontiertech.com
Bob Natale	natale@acec.com
Scott Neal	scott_neal@hp0800.desk.hp.com
Bill Norton	wbn@merit.edu
Barbara Packard	bpackard@ppg01.sc.hp.com
Sudhir Pendse	sudhir@netcom.com
Dave Perkins	dperkins@synoptics.com
Eric Peterson	ericpe@microsoft.com
Mary Quinn	mquinn@ftp.com
Marshall T. Rose	mrose@dbc.mtview.ca.us
Rick Segal	rsegal@microsoft.com
Reuben Sivan	rds@world.std.com
Simon E. Spero	ses@tipper.oit.unc.edu
Bob Stewart	rlstewart@eng.xyplex.com
Wayne F. Tackabury	wayne@cayman.com
Ling Thio	H.L.Thio@et.tudelft.nl
Mark Towfiq	towfiq@sunsite.unc.edu
Chuck Townsend	townsend@ctron.com
Alex Tudor	alex@hpssdat.sc.hp.com
Chuck Wegrzyn	wegrzyn@nic.cerf.net
Pete Wilson	pwilson@world.std.com
Boris Yanovsky	boris@netmanage.com
Chris Young	cyoung@ctron.com
Dennis Young	young@telebit.com

2. PROGRAMMING WITH WINDOWS SNMP

This section outlines some of the high level considerations relevant to the programming "model" envisioned by WinSNMP. This model is meant to add background and context for evaluating the specification itself. In general, although it is not possible to eschew all references to implementation details in the MS-Windows environments, the WinSNMP specifications try to openly state any and all relevant implementation assumptions.

The primary assumption is that the WinSNMP/Manager API will be implemented by (or for) the SNMP service provider as a dynamic link library (WINSNMP.DLL). This DLL might perform the SNMP functions locally or might be a “helper” DLL that ships application requests to an SNMP service on a remote platform and, in return, routes responses from that platform back to the target applications on the local MS-Windows machine.

In either case, the major aspects of WinSNMP implementation that affect application development include:

- Levels of SNMP Support
- Transport Interface Support
- Entity/Context Translation Modes
- Local Database Information
- Session Characteristics
- Memory Management
- Asynchronous Model
- Polling and Retransmission
- Error Handling
- Data Types

2.1. Levels of SNMP Support

This specification allows for multiple levels of SNMP support--explicitly for implementations and implicitly for applications.

Note that these "Levels" are independent of and unrelated to the "Modes" of interpretation of entity and context arguments (described later).

Note that the implementation will report its maximum level of SNMP support in response to the **SnmpStartup** function (described later).

2.1.1. Implementations

The WinSNMP API specification allows an ***implementation*** to support any of four overlapping levels of SNMP operations:

- **Level 0** = Message encoding/decoding only
- **Level 1** = Level 0 + interaction with SNMPv1 agents
- **Level 2** = Level 1 + interaction with SNMPv2 agents
- **Level 3** = Level 2 + interaction with other SNMPv2 managers

2.1.1.1. "Level 0" Implementations

"Level 0" implementations must support all WinSNMP specifications ***except*** those which require communication with other SNMP entities, namely:

- **SnmpSendMsg**
- **SnmpRecvMsg**
- **SnmpRegister**

"Level 0" implementations exist to provide SNMP message encoding and decoding services to applications which do not require the communications transport services of the WinSNMP implementation, but still require WinSNMP services, such as:

- Local Database Functions
- **SnmpEncodeMsg**
- **SnmpDecodeMsg**

All WinSNMP implementations ***must*** include full "Level 0" support.

2.1.1.2. "Level 1" Implementations

"Level 1" implementations support communications with SNMPv1 agents, in addition to providing full "Level 0" support.

Since WinSNMP applications are ***structured*** to support SNMPv2, "Level 1" implementations ***must*** support the requisite transformations specified in the "Coexistence" document (RFC1452). For example, if a WinSNMP application submits a GetBulkRequest PDU to a "Level 1" implementation, the WinSNMP implementation will transform this into a GetNextRequest PDU, per Section 3.1.1.(2) of RFC1452, and proceed accordingly.

Note that WinSNMP always returns traps in SNMPv2 format, whether the trap emanates from an SNMPv1 agent or, as a notification, from an SNMPv2 agent. This behavior is also defined by RFC1452.

"Level 1" implementations ***must*** support the use of target agent addresses and community strings; but are ***not required*** to support any SNMPv2 mechanisms, other than the "Coexistence" transformations mentioned above.

2.1.1.3. "Level 2" Implementations

"Level 2" implementations support communications with SNMPv2 agents, in addition to providing full "Level 1" and "Level 0" support.

In particular, "Level 2" implementations ***must*** support the Party MIB and the use of parties, contexts, authentication, and privacy mechanisms, but are ***not required*** to support the Manager-to-Manager MIB or protocol operations relating to the InformRequest PDU type.

2.1.1.4. "Level 3" Implementations

"Level 3" implementations support communications with other SNMPv2 management entities via the Manager-to-Manager MIB and protocol operations relating to the InformRequest PDU type, in addition to providing full "Level 2", "Level 1", and "Level 0" support.

2.1.2. Applications

The WinSNMP API is oriented toward the writing of ***applications*** which are SNMPv2-enabled, at least in terms of their structure. A WinSNMP application may always use the relevant PDU types defined for SNMPv2 (as specified in WinSNMP.h, the "Declarations" section of this document) with the assurance that the implementation will perform the necessary transformations--in accordance with the "Coexistence" document (RFC1452)--when communicating with an SNMPv1 agent on behalf of the application. Likewise, a WinSNMP application will always receive Trap PDUs (via **SnmpRecvMsg** from the implementation) as SNMPv2 traps, even when the issuing entity is an SNMPv1 agent.

Note that it is possible for WinSNMP applications to operate in such a way as to utilize the implementation merely for SNMP message encoding and decoding and to bypass the WinSNMP implementation with respect to communications with the destination entities. In this mode, the application must perform the necessary GetResponse and Trap PDU transformations for itself, at its own discretion.

2.2. Transport Interface Support

For everything above “Level 0”, the WinSNMP implementation conducts the communications transactions with the SNMP agents on behalf of the applications. Nothing in the WinSNMP specification attempts to dictate how an implementation (and/or an application) will actually execute the communications process with remote entities.

A number of options exist. They are not necessarily mutually exclusive--several might be used by an implementation with one or more in the same or a different combination being used by its client applications.

Possible approaches include the following: Embedded Stack, Proprietary Stack API, Windows Sockets API, Multi-Protocol API, RPC.

In the “embedded stack” approach, the implementation incorporates the transport layer, without overt reliance on any external components. Note that in this context “embedded” simply means that it is part of the WinSNMP implementation package and “external” means “not provided by the WinSNMP provider”. In other words, an “embedded” transport could actually reside in a separate physical module (e.g., a DLL or virtual driver (VxD) of its own).

Using the “proprietary stack API” approach, a WinSNMP implementation supports the development API of a transport stack vendor. Such a technique might be used for competitive, marketing, or performance reasons (among others). This approach can often yield access to low-level protocol elements that are sometimes not available in the vendor-independent and multi-protocol options.

The “Windows Sockets API” (WinSock) approach affords an implementation--and, consequently its users--a very comfortable degree of independence and flexibility. WinSock is fast becoming an industry standard for TCP/IP applications programming. It enables stack vendors to exploit their proprietary APIs on the back-end while offering application developers a single interface on the front-end. The WinSock approach is the one that is assumed (but not required) by the WinSNMP API specifications, as shown in Figure 1.

The “Multi-Protocol API” approach allows an implementation to include support for a diverse set of transport interfaces. The APP2SOCK.DLL supplied by Spry is an example of such an API. Enhanced flexibility is the main advantage of this approach, while limited support for some of the lower-level details of the underlying protocols and potential performance impacts are possible disadvantages. It is anticipated that v2 of the WinSock API will include support for multiple transport protocols (including TCP/IP, of course).

Finally, the RPC approach permits implementations which function only as “mediators” between applications on the local MS-Windows desktop and a remote SNMP service provider on, for example, a UNIX platform.

The main point of this survey of available communications strategies is that there are options; they are not necessarily mutually exclusive; they can be mixed and matched across both WinSNMP implementations and applications.

2.3. Entity/Context Translation Modes

WinSNMP applications have the capability of instructing the implementation to interpret entity and context arguments as either literal SNMPv1 agent address and community string, respectively, or as literal SNMPv2 party and context IDs, respectively. An alternative to either of these modes is that in which these arguments are interpreted as user- or application-friendly names for entities and managed object collections to be dereferenced ("translated") into their respective SNMPv1 or SNMPv2 components via the implementation's local database.

The three entity/context translation modes are:

```
SNMPAPI_TRANSLATED          = Translate via Local Database look-up
SNMPAPI_UNTRANSLATED_V1 = Literal transport address and community string
SNMPAPI_UNTRANSLATED_V2 = Literal SNMPv2 party and context IDs
```

The WinSNMP implementation will always identify its current default entity/context translation mode setting in the return value from the **SnmpStartup** function (which is idempotent). A WinSNMP application may request a different entity/context translation mode setting at any time with the **SnmpSetTranslateMode** function. An implementation may elect to not support a requested translation mode, in which case it must return an error and set **SnmpGetLastError** to SNMPAPI_MODE_INVALID.

All WinSNMP implementations and applications are encouraged to support SNMPAPI_TRANSLATED mode as their default mode of operation, to assist in fostering bilingual (SNMPv1 and SNMPv2) applications deployment.

Note that the sample code which follows in sections 2.3.1 through 2.3.4 includes literal string representations of some of the arguments to the WinSNMP functions. This is merely for expository purposes. In the interests of internationalization and localization--and generally good software engineering practices--application writers are encouraged to isolate all such text string values in StringTables in separate resource files or to use some similar technique to modularize such strings out of the operating logic of their applications.

Also, note that context “string” arguments are passed as “octet string” structures (smiOCTETS descriptors) since SNMPv1 “community strings” can contain any values, not just those from the NVT ASCII or “DisplayString” character set.

2.3.1. SNMPAPI_TRANSLATED Mode

When the translation mode is set to SNMPAPI_TRANSLATED, an application will make calls similar to the following:

```
LPCSTR entityName = “Main_Hub”;
smiOCTETS contextName;
contextName.ptr = “Traffic_Stats”;
contextName.len = lstrlen (contextName.ptr);
hAgent = SnmpStrToEntity (hSomeSessin, entityName);
hView   = SnmpStrToContext (hSomeSession, const &contextName);
```

The implementation will use its selected access method to look-up "Main_Hub" and "Traffic_Stats" in its local database and, if successful, will assemble the appropriate internal data structures and return HANDLE values for use by the application.

2.3.2. SNMPAPI_UNTRANSLATED_V1 Mode

When the translation mode is set to SNMPAPI_UNTRANSLATED_V1, an application will make calls similar to the following:

```
LPCSTR entityName = “192.151.207.34”;
smiOCTETS contextName;
contextName.ptr = “public”;
contextName.len = lstrlen (contextName.ptr);
hAgent = SnmpStrToEntity (hSomeSessin, entityName);
hView   = SnmpStrToContext (hSomeSession, const &contextName);
```

The implementation will assume--based on the SNMPAPI_UNTRANSLATED_V1 setting for hSomeSession--that "192.151.207.34" equates to an IP address to be reached via UDP port 161, and that this value is being passed as a far pointer to a constant NULL terminated text string (LPCSTR) that it must first convert to dotted decimal notation.

2.3.3. SNMPAPI_UNTRANSLATED_V2 Mode

When the translation mode is set to SNMPAPI_UNTRANSLATED_V2, an application will make calls similar to the following:

```
LPCSTR entityName = “1.3.6.1.6.3.3.1.3.134.141.40.162.1”;
smiOCTETS contextName;
contextName.ptr = “1.3.6.1.6.3.3.1.4.134.141.40.162.1”;
contextName.len = lstrlen (contextName.ptr);
hAgent = SnmpStrToEntity (hSomeSessin, entityName);
hView   = SnmpStrToContext (hSomeSession, const &contextName);
```

The first string is an initialPartyID with an agent from 134.141.40.162; the second string is an intialContextID with the same agent.

The implementation will assume--based on the SNMPAPI_UNTRANSLATED_V2 setting for hSomeSession--that "1.3.6.1.6.3.3.1.3.134.141.40.162.1" equates to an a PartyID instance at IP address 134.141.40.162 to be reached via UDP port 161, and that this value is being passed as a far pointer to a constant NULL terminated text string (LPCSTR) that it must first convert to an OID.

2.4. Local Database

An SNMPv1 message includes version information and a community string, in addition to the PDU. An SNMPv2 message includes source party, destination party, context, and authentication information, in addition to the PDU (and the entire message may optionally be encrypted). Given these facts and the stated mission to accommodate both existing versions of SNMP, WinSNMP must meet at least the following four objectives:

1. A WinSNMP application must have full access to all components of the SNMP message issued by the WinSNMP implementation. At the extreme, the **SnmpEncodeMsg** and **SnmpDecodeMsg** functions enable access to and manipulation of fully-serialized, "ready-for-transport" SNMP messages.

2. A WinSNMP application must not have to incorporate WinSNMP implementation-specific routines or data structures to utilize any of the functionality defined by WinSNMP itself. Each WinSNMP implementation may use private mechanisms external to the WinSNMP applications, but any and all ***necessary*** interfaces to these mechanisms will be via the defined WinSNMP APIs only.

3. A WinSNMP application must not have to know the SNMP version level of the target SNMP entities acting in an agent role. The WinSNMP implementation will perform any and all necessary mappings between SNMPv1 and SNMPv2 in accordance with the appropriate RFCs, and especially RFC 1452. With respect to agent addressing, this is particular true for “TRANSLATED” mode access; for protocol operations it holds regardless of the entity/context translation mode in effect.

4. One implication of the foregoing requirement is that the SNMPv1 message format must fit neatly within the structure adopted for the SNMPv2 message format. This statement applies to WinSNMP "messages" only--it is not meant in any way to limit or modify anything in the "Coexistence" RFC.

Taking the view that SNMPv1 message semantics can be housed within SNMPv2 message semantics, we will first focus on the SNMPv2 message components:

For SNMPv2 communications, the "source party" (srcEntity) components refer to the management application and will largely be supplied by the WinSNMP implementation layer via the Local Database. For SNMPv1 communications, srcEntity basically is a no-op.

For SNMPv2 communications, the "destination party" (dstEntity) components refer to the target agent and must come, at least in part, from logically remote SNMP entities. Given a dstEntity transport address and protocol and the assumption of at least one noAuth/noPriv "entrance" into the target agent (i.e., InitialPartyID), an application can initiate SNMP exchanges via the WinSNMP implementation. For SNMPv1 communications, dstEntity refers to the transport address of the target agent entity.

For SNMPv2 communications, the "context" identifies a collection of managed object resources accessible to a management application under the control of the target agent. For SNMPv1 communications, the context parameter refers to "community string".

A driving force behind the approach taken in this specification with respect to these issues is the need to accommodate SNMPv2 administrative and protocol constructs in a symmetrical fashion, and at the same time transparently subsume SNMPv1 administrative and protocol constructs.

The major assumption is that the implementation’s "local database"--at least for “TRANSLATED” mode operations--contains source party, destination party, and context entries (and possibly other data). Consequently, the **SnmpSendMsg** function does not need to include "qos", "timeout", "retry", or similar values since these are available in the Local Database.

2.5. Sessions

The "session" created by the **SnmpOpen** function is used to manage the link between the WinSNMP application and the WinSNMP interface implementation. That is, the session is the unit of resource and communications management between a calling WinSNMP application and its supporting WinSNMP implementation. A well-behaved WinSNMP application will use the session construct to logically organize its operations and to minimize resource requirements on the implementation. The following statements summarize the role and certain attributes of WinSNMP sessions:

- A "session" is opened with **SnmpOpen**, and closed with **SnmpClose**.
- A "session-id" is returned by the **SnmpOpen** function to the application as a HANDLE variable, which the implementation may use internally to manage resources.
- An application can have multiple sessions open at one time, subject to the limitations stated below.
- The minimum number of concurrent sessions which an implementation must support is one.
- The maximum is undefined and is implementation-specific and, possibly, resource-dependent.
- When an application's request to open a session cannot be granted because of the limitations stated above, the implementation will return SNMPAPI_FAILURE to **SnmpOpen** and will set **SnmpGetLastError** to report SNMPAPI_ALLOC_ERROR.
- All WinSNMP API functions--except **SnmpOpen**--which return HANDLE variables will include a "session-id" handle as an input parameter, so that the implementation can use it internally to manage and account for resources on behalf of the session.
- HANDLE variables created under one open session can be utilized by other open sessions (if any) ***within*** a given application (task). Optionally, an implementation may internally share HANDLE variables among sessions in separate applications. Note this optional resource efficiency, if it is supported by an implementation, is totally transparent to the application.
- When an application closes a session by executing the **SnmpClose** function, all resources created on behalf of that session by the implementation, and not previously freed by the application, will be freed automatically by the implementation. If an implementation supports the optional sharing of HANDLE variables among open sessions across multiple applications, then the resources will not be physically freed until the final open session which "created" the resources closes.
- Sessions may have other attributes, above and beyond those discussed above (e.g., the 'dstEntity' and 'context' interpretation modes of TRANSLATED, UNTRANSLATED_V1, and UNTRANSLATED_V2).

2.6. Memory Management

The allocation, ownership, deallocation, and “garbage collection” of memory objects is often a troublesome issue in a complex multi-provider MS-Windows programming arrangement. It really is not a question of adequate capabilities being provided by the environment. It is a question of understanding the options and the rules, agreeing to a division of labor, authority, and responsibility among the components; and, finally, competence and diligence in implementing such an agreement.

In MS-Windows programming, it is important to remember that the implementation, as a DLL, is actually just an extension of the calling application. Applications can allocate, use, and deallocate memory; if they terminate without freeing allocated memory, MS-Windows deallocates it for them automatically. If a DLL allocates memory (without taking explicit actions to the contrary by declaring the GMEM_SHARE option), then it is actually “owned” by the currently connected application and is identical to memory allocated directly by the application. A DLL can also deallocate application-owned memory on behalf of the calling application. A DLL can invoke the GMEM_SHARE option (not recommended if portability to Windows-NT is desired) to allocate memory that it will “own”, and/or it can allocate memory out of its local dataspace to provide for “persistent objects” that might be shared among diverse applications.

The WinSNMP “arrangement” includes three different kinds of memory "objects":

- HANDLE'd resources
- C-style (NULL terminated) strings
- Non-scalar WinSNMP API data types of variable length

2.6.1. HANDLE’d Resources

There are five varieties of HANDLE'd resources:

- Sessions
- Entities
- Contexts
- Protocol Data Units (PDUs)
- VarBindLists (VBLs)

These objects are accessed via handles for two reasons:

- To hide their structures from the applications; and

- to permit implementations to optimize and/or differentiate themselves vis-à-vis their construction and manipulation of these objects "behind" the API.

All HANDLE'd objects are of data type “HSNMP_<object_tag>“ and are always "owned" by the implementation. An application may request their creation and may signal their eligibility for deletion and reclamation, but these operations (like all others concerning these objects) are indirect...the realization is up to the implementation.

An implementation that wants to permit sharing of HANDLE’d resources (most likely) allocates them out of its local data space. If it wanted to restrict them to the calling application’s scope, it would allocate them out of global memory without the GMEM_SHARE option. Both approaches can be employed in an implementation -- for example, it might make sense to share Entity and Context objects across multiple applications, but it is less likely that Sessions, PDUs, and VarBindLists would benefit significantly from such treatment.

2.6.2. C-Style Strings

The C-style (NULL terminated) strings are provided mainly for convenience to easily convert Entity and OID objects to and from the most common string representation. The WinSNMP functions which use C-style strings are limited to: **SnmpStrToEntity**, **SnmpEntityToStr**, **SnmpStrToOid**, and **SnmpOidToStr**. (The inclusion of "Str" in the name is a bit misleading in the case of the **SnmpStrToContext** and **SnmpContextToString** functions, as the "context" parameter in these functions must be an SNMP-style "octet string" to accommodate the legal data values.)

The application is entirely responsible for allocating, managing, and freeing this memory, as might be appropriate to its specific operating requirements and/or circumstances. This will require passing a "size" parameter to the implementation in functions which use pointers to C-style string variables as output arguments (i.e., **SnmpEntityToStr** and **SnmpOidToStr**).

2.6.3. Descriptors

There are three non-scalar WinSNMP API data types, of variable length:

- smiOCTETS
- smiOID
- smiVALUE.

All three are structures. The first two are both "descriptor" structures, consisting of two members: "len" and "ptr". For smiOCTETS, "len" is an unsigned long integer (smiUINT32) value indicating the number of bytes in the subject octet string (no necessary NULL terminating byte) and "ptr" is a far pointer to a byte array containing the octet string. For smiOID, "len" is an unsigned long integer value indicating the number of unsigned long integers in the subject OID and "ptr" is a far pointer to an array of unsigned long integers representing the OID's sub-identifiers.

The smiVALUE structure is different and a bit more complex. It too consists of just two members. The first is an unsigned long integer indicating the "syntax" of the second member. The second member is the union of all the possible WinSNMP API data types. A calling application must first check the "syntax" member of a returned smiVALUE structure to know how to dereference the second member, which might be a simple scalar value or might be one of the WinSNMP API structures with defined syntax (including an smiOCTETS, or one of its derivatives such as smiIPADDR, or an smiOID). In general, this is pretty typical SNMP API fare. In actuality, the smiVALUE structure is not a problem--it is always of a fixed size.

It is only when its "syntax" member indicates that the "value" member is either an smiOCTETS or an smiOID structure (which contain pointers to variable length data) that the memory management "agreement" becomes important. Specifically, who assigns the pointers (i.e., allocates the memory), who fills in the "len" members, who owns these objects, and who is responsible for freeing the resources when they are no longer needed or in cases of memory shortage?

Fortunately, the statement of this problem is more complex than the statement of its resolution!

- For input parameters, the application provides the structure and populates its members (i.e., allocates the memory for the variable length objects).

- For output parameters, the application again provides the structure, but the implementation populates its members (i.e., allocates the memory for the variable length objects).

- The application must use an appropriate functions (e.g., GlobalFreePtr) to free the memory that it has allocated for such input parameters and must use the **SnmpFreeDescriptor** WinSNMP function to free the memory allocated by the implementation for these output parameters.

See Section 2.11.4. Descriptors, in Section 2.11. WinSNMP Data Types.

The combined effects of this particular “agreement” yield substantial benefits:

- It clearly delineates a small number of cooperative memory management requirements.
- It clearly assigns responsibility in each case.
- It reduces the likelihood of over-allocation of temporary buffer space.
- It reduces the likelihood of unnecessary buffer copying (from max-sized temporary buffers to right-sized working buffers).
- It leverages the “natural” memory management posture of MS-Windows while providing independent flexibility in this area to both applications and implementations alike.

2.6.4. Summary

As a general rule, the WinSNMP application is responsible for freeing all WinSNMP resources allocated via calls to the WinSNMP API using the following functions:

- **SnmpFree**<xxx>: Entity, Context, Pdu, Vbl, Descriptor
- **SnmpClose**: Session
- **SnmpCleanup**: Task

These calls are 'cumulative', in the sense that **SnmpFree**<xxx> frees a single specific HANDLE'd resource; **SnmpClose** frees all such resources allocated to a given session within the calling task, and the session HANDLE'd resource itself; **SnmpCleanup** performs (in effect) an **SnmpClose** on all open sessions within the task. Applications are encouraged to use these in the order shown as appropriate to the application’s processing logic.

It is also very important to note that ***every time*** a WinSNMP function calls results in the return of a HANDLE’d object resource to the application, each and every such resource will be a ***new*** resource. In this context, “new” means a unique value for that kind of resource at that instant in the calling application. This means, for example, that it is safe--with respect to a subsequent **SnmpRecvMsg** call--to free the srcEntity, dstEntity, context, and pdu resources right after calling **SnmpSendMsg**, But a given app might need or want to retain them, of course. If you built these resources for, say, a polling operation, then you will probably want to retain them for the next iteration...and in many cases you will want to match up out-bound RequestIDs with received RequestIDs.

An allocated HANDLE’d object resource is never freed by the implementation except upon request by the application using one of the three types of calls outlined at the top of this sub-section or--as an optional capability of an implementation--upon ungraceful termination of an application which otherwise left resources allocated. It is the application’s responsibility to request the creation of and the deletion of all WinSNMP HANDLE’d object resources. The implementation is free to perform these operations any way it wants internally, as long as the external appearance to the application accords with this set of specifications.

2.7. Asynchronous Model

One contemporary programming model has applications "driven" by the receipt and processing of asynchronous message-events. This asynchronous message-driven model maps well to modern object-oriented theory, the SNMP distributed management paradigm, and the Microsoft Windows programming and runtime environments. Likewise, although WinSNMP does not presume any particular transport mechanism for the conveyance of SNMP messages between managers and agents, it is to be noted that, fundamentally, SNMP is a datagram-based protocol, in which no actual channel (virtual circuit) is established between remote entities. This behavior also maps well to the message-driven programming model. For those reasons, among others, this is the programming model adopted by WinSNMP.

Modern message-driven applications typically must respond to other kinds of important events, some of which may rely on synchronous relationships. Actually, all of the functions specified in the WinSNMP API have a synchronous component--most are totally synchronous; three critical ones have an asynchronous dimension:

- **SnmpSendMsg**
- **SnmpRecvMsg**
- **SnmpRegister**

Of these, **SnmpRecvMsg** has the most impact on asynchronous operations.

The basic asynchronous model for programming with WinSNMP follows these steps:

1. The application opens a session with the WinSNMP implementation (with the **SnmpOpen** function).

2. If the application is interested in receiving traps, it indicates this (with the **SnmpRegister** function).

3. The application prepares one or more PDUs for transmission to and processing by the WinSNMP implementation via WinSNMP "messages" (using **SnmpCreatePdu** and other PDU, Variable-Binding, and Utility functions).

4. The application submits one or more asynchronous requests consisting of an SNMP PDU and message "wrapper" elements (with the **SnmpSendMsg** function).

5. The application receives notification that a response to a request is available or that a registered trap has occurred (via the message "channel" specified in the **SnmpOpen** function).

6. The application retrieves the response (with the **SnmpRecvMsg** function).

7. The application processes the response as appropriate (using application-specific logic).

8. The application closes the WinSNMP session (with the **SnmpClose** function).

Note that, in general, steps 2 through 7 can take place in nearly any order and at any time during program execution.

2.8. Polling and Retransmission

Given the asynchronous nature of both SNMP itself and the WinSNMP **SnmpSendMsg**, **SnmpRecvMsg**, and **SnmpRegister** functions, users of this specification (i.e., implementors and applications writers) must be concerned with timeout and retry issues. Taken together, timeout and retry will be referred to hereinafter as "retransmission". (Note that no "back-off" mechanisms are currently included.)

Applications have sole responsibility for ***polling***: establishing the frequency, initiating transactions, and timer management, among other things. This ensures that applications have knowledge of the “request-id” component of the out-going PDUs.

With respect to ***retransmission***, applications clearly have the primary responsibility, regarding both policy and execution. Implementations ***must*** provide retransmission policy support (via their local database) and ***may*** optionally provide retransmission execution support.

Accordingly, in WinSNMP applications the timeout period, in practice, refers to the elapsed time between an application's issuance of an **SnmpSendMsg** request and receipt of the corresponding message via the **SnmpRecvMsg** function. From the perspective of the implementation, the timeout period will refer to the elapsed time between the actual sending of an SNMP request message to a destination entity and the receipt of the SNMP response message from that destination.

The fundamental retransmission policy mechanism will be the Local Database. Each potential destination entity entry in the Local Database will include--among other attributes--timeout (elapsed time in seconds) and retry (count) elements. These values can be stored in and retrieved from the Local Database by an application with the **Snmp[Get|Set]Timeout** and **Snmp[Get/Set]Retry** functions. At runtime, an application may elect to use, update, or ignore the default values in the Local Database. When an implementation which supports retransmit execution is operating in retransmit mode, it ***must*** use the timeout and retry values from the Local Database for the respective destination entities.

Note that none of the foregoing precludes or impedes the "out-of-the-box" mode of operation. An implementation can (and should) "boot up" with some generic default values in its (conceptual) Local Database for use when an application initializes entities in the SNMPAPI_UNTRANSLATED_V[1|2] modes.

So, for WinSNMP, the following summarizes the timeout/retry approach:

- The application manages the policy via the Local Database functions by storing “desired” values for each destination entity. Optionally, the implementation ***may*** also update the “actual” observed values in its local database for subsequent use by the application in adjusting the “desired” (policy) values..

- The application executes the policy, at its discretion. That is, when it issues a request (via **SnmpSendMsg**) and wants to monitor the time-out event, it sets a timer (most likely using the “desired” time-out value retrieved from the local database).

- If the response comes in before the timer goes off, it cancels the timer and that's the end of it. If the timer expires, it decides then whether to retry (most likely, but not only, based on the retry count value retrieved from the local database).

- If, during the course of execution, the application determines that either the default time-out and/or retry values are inappropriate in can either ignore that fact, or change its runtime behavior accordingly, and/or modify the default values for the respective entities in the Local Database.

- It may well be, given the above, that certain "network smart" apps might populate and update the default values in the Local Database, while many more "network agnostic" applications just use the default values, whether just for its policy (when the implementation actually does the execution) or for both policy and execution purposes.

- Applications may request that the implementation execute the retransmission policy (using the values in the Local Database) via the **SnmpSetRetransmitMode** function, with (SNMPAPI_ON). A valid response to this request by a compliant implementation is either SNMPAPI_SUCCESS or SNMPAPI_MODE_INVALID.

- In the former case, the application may elect to leave retransmission execution entirely to the implementation or to augment it with its own execution, if desired. An application can use **SnmpSetRetransmit** again, with (SNMPAPI_OFF), to "turn off' the implementation in this regard.

- When the implementation executes the retransmission policy, it repeats the original request-id component in each retransmitted PDU.

- When the implementation responds to the **SnmpSetRetransmitMode** (SNMPAPI_ON) request with the SNMPAPI_MODE_INVALID error, the application must assume all responsibility for execution of the retransmission policy.

- A "standard" set of timer support functions for use by WinSNMP applications developers might be added to the WinSNMP specification at a future date.

2.9. RequestIDs

The original release of the v1.1 specification indicated that the **SnmpSendMsg** and **SnmpRecvMsg** functions returned the RequestID of the associated PDU. This proved to be an unworkable mechanism due to the need to accept a zero-valued RequestID as a legitimate value. This v1.1a release corrects that error, to wit:

- RequestID of 0 must be allowed in PDUs, ergo:
- WinSNMP API functions cannot use RequestID as a return value, since SNMPAPI_FAILURE == 0; ergo:
- **SnmpSendMsg** and **SnmpRecvMsg** will return either SNMPAPI_SUCCESS [1] or SNMPAPI_FAILURE [0], ***not*** the RequestID.
- Applications can use the **SnmpGetPduData** function to retrieve a PDU's RequestID when necessary.
- An application can ask the implementation to generate and assign a RequestID to a PDU by passing zero in the RequestID parameter of the **SnmpCreatePdu** function and can then determine that value, if desired with the **SnmpGetPduData** function, per above.
- The implementation will use its best efforts to generate RequestIDs which are temporally unique internally (i.e., a simple incrementing algorithm will suffice). It does not have to avoid conflicts with externally generated RequestIDs. It may assign zero as a generated RequestID in the normal course of events.
- An application that wants to force a RequestID of zero (or any other value) can use the **SnmpSetPduData** function to do so.

This correction required corresponding changes to the descriptive text (only) of the **SnmpSendMsg** and **SnmpRecvMsg** functions to specify the correct return values and to the **SnmpCreatePdu** function to explain the effect of passing RequestID=0 in this function, per above.

Note that, overall, this approach is somewhat less attractive from the application's point of view, since automatic RequestID generation/assignment is a one-short deal per PDU in this scenario. On the other hand, it is likely that most apps will want to generate their own RequestIDs (not hard and simplifies things) or just will not care (i.e., use the Session parameters to distinguish response channels anyway).

2.10. Error Handling

All WinSNMP functions have an immediate return value. If this value is SNMPAPI_FAILURE (0), it means that the implementation detected or encountered an error of some kind. The application must then call the **SnmpGetLastError** function to retrieve the extended error information describing the specific problem encountered.

The bifurcation into "common" and "context-specific" error codes in this section serves merely to allow an abbreviation of error condition descriptions in the function reference sections. The distinction between "SNMP error codes" and "SNMP API error codes" in the "context-specific" section is somewhat more significant. The former are fixed by the RFCs; the latter are creations of this specification.

2.10.1. Common Error Codes

Any WinSNMP function can fail with any one of the following error codes returned via **SnmpGetLastError**:

SNMPAPI_NOT_INITIALZED
SNMPAPI_ALLOC_ERROR
SNMPAPI_OTHER_ERROR

SNMPAPI_NOT_INITIALIZED signals that **SnmpStartup** was not successfully executed, either since program execution began or since **SnmpCleanup** successfully completed. Note that if **SnmpStartup** fails, an immediate call to **SnmpGetLastError** (i.e., before any other WinSNMP calls) will return the error code applicable to the failure of **SnmpStartup**; all subsequent calls to WinSNMP functions before a successful **SnmpStartup** execution will fail with SNMPAPI_NOT_INITIALIZED.

SNMPAPI_ALLOC_ERROR signals that the implementation was unable to obtain sufficient resources to carry out the requested action. Applications should respond by freeing resources, or by reducing the resource requirements of the request, or by informing the user (e.g., via MessageBox or log file entry) and facilitating a graceful shutdown via **SnmpClose** calls and/or **SnmpCleanup**.

SNMPAPI_OTHER_ERROR signals an unknown, undefined, or otherwise indeterminate error occurred. Implementations may provide an optional, ancillary, and independent means of providing additional feedback to the user for subsequent problem resolution. In most cases, applications should attempt to shutdown gracefully via **SnmpClose** calls and/or **SnmpCleanup** after receiving this error.

Since each of these error conditions could arise on each and any of the WinSNMP API functions, they are documented here only. The error information section of each function description refers to these collectively as "Common Error Codes" and documents any other error conditions which might be specific to the given function.

2.10.2. Context-Specific Error Codes

The following lists are excerpted from the "Declarations" section of this document (which essentially constitutes the WinSNMP.h include file). They are included here mainly as a place-holder for a future elaboration of each error condition, similar to what was done in the preceding section for "Common Error Codes".

```
/* Syntax Values for Exception Conditions in SNMPv2 Response Varbinds */
#define SNMP_VALUE_NOSUCHOBJECT       (ASN_CONTEXT | ASN_PRIMITIVE | 0x0)
#define SNMP_VALUE_NOSUCHINSTANCE     (ASN_CONTEXT | ASN_PRIMITIVE | 0x1)
#define SNMP_VALUE_ENDOFMIBVIEW       (ASN_CONTEXT | ASN_PRIMITIVE | 0x2)

/* SNMP Error Codes Returned in Error_status Field of PDU...Not API Error Codes */
/* Error Codes Common to SNMPv1 and SNMPv2 */
#define SNMP_ERROR_NOERROR                      0
#define SNMP_ERROR_TOOBIG                       1
#define SNMP_ERROR_NOSUCHNAME                   2
#define SNMP_ERROR_BADVALUE                     3
#define SNMP_ERROR_READONLY                     4
#define SNMP_ERROR_GENERR                       5
/* Error Codes Added for SNMPv2 */
#define SNMP_ERROR_NOACCESS                     6
#define SNMP_ERROR_WRONGTYPE                    7
#define SNMP_ERROR_WRONGLENGTH                  8
#define SNMP_ERROR_WRONGENCODING                9
#define SNMP_ERROR_WRONGVALUE                  10
#define SNMP_ERROR_NOCREATION                  11
#define SNMP_ERROR_INCONSISTENTVALUE           12
#define SNMP_ERROR_RESOURCEUNAVAILABLE         13
#define SNMP_ERROR_COMMITFAILED                14
#define SNMP_ERROR_UNDOFAILED                  15
#define SNMP_ERROR_AUTHORIZATIONERROR          16
#define SNMP_ERROR_NOTWRITABLE                 17
#define SNMP_ERROR_INCONSISTENTNAME            18

/* WinSNMP API Function Return Codes */
#define SNMPAPI_FAILURE                 0       /* Generic error code */
#define SNMPAPI_SUCCESS                 1       /* Generic success code */
/* WinSNMP API Error Codes (for SnmpGetLastError) */
#define SNMPAPI_ALLOC_ERROR             2       /* Error allocating memory */
#define SNMPAPI_CONTEXT_INVALID         3       /* Invalid context parameter */
#define SNMPAPI_CONTEXT_UNKNOWN         4       /* Unknown context parameter */
#define SNMPAPI_ENTITY_INVALID          5       /* Invalid entity parameter */
#define SNMPAPI_ENTITY_UNKNOWN          6       /* Unknown entity parameter */
#define SNMPAPI_INDEX_INVALID           7       /* Invalid VBL index parameter */
#define SNMPAPI_NOOP                    8       /* No operation performed */
#define SNMPAPI_OID_INVALID             9       /* Invalid OID parameter */
#define SNMPAPI_OPERATION_INVALID       10      /* Invalid/unsupported operation */
#define SNMPAPI_OUTPUT_TRUNCATED        11      /* Insufficient output buf len */
#define SNMPAPI_PDU_INVALID             12      /* Invalid PDU parameter */
#define SNMPAPI_SESSION_INVALID         13      /* Invalid session parameter */
#define SNMPAPI_SYNTAX_INVALID          14      /* Invalid syntax in smiVALUE */
#define SNMPAPI_VBL_INVALID             15      /* Invalid VBL parameter */
#define SNMPAPI_MODE_INVALID            16      /* Invalid mode parameter */
#define SNMPAPI_SIZE_INVALID            17      /* Invalid size/length parameter */
#define SNMPAPI_NOT_INITIALIZED         18      /* SnmpStartup failed/not called */
#define SNMPAPI_MESSAGE_INVALID         19      /* Invalid SNMP message format */
#define SNMPAPI_HWND_INVALID            20      /* Invalid Window handle */
/* Others will be added as needed */
#define SNMPAPI_OTHER_ERROR             99      /* For internal/undefined errors */
```

2.10.3. Transport Error Reporting

In the case of errors which are detected at the time of accepting a request to send or receive a packet, these are returned synchronously by **SnmpSendMsg**, **SnmpRecvMsg**, or **SnmpRegister** via a return code of SNMPAPI_FAILURE (which the application must follow-up with a call to **SnmpGetLastError** (to retrieve the extended error code). In the case of errors which are detected after the packet has gone out onto the wire, the WinSNMP implementation sends a packet receipt notification to the affected session and these errors are returned via an SNMPAPI_FAILURE indication from the next **SnmpRecvMsg** call on that session.

The generic transport layer (TL) error codes for the WinSNMP/Manager API are:

```
#define SNMPAPI_TL_NOT_INITIALIZED      100   /* Transport layer not initialized */
#define SNMPAPI_TL_NOT_SUPPORTED        101   /* Transport does not support protocol */
#define SNMPAPI_TL_NOT_AVAILABLE        102   /* Network subsystem has failed */
#define SNMPAPI_TL_RESOURCE_ERROR       103   /* Transport resource error */
#define SNMPAPI_TL_UNDELIVERABLE        104   /* Destination unreachable */
#define SNMPAPI_TL_SRC_INVALID          105   /* Source endpoint invalid */
#define SNMPAPI_TL_INVALID_PARAM        106   /* Input parameter invalid */
#define SNMPAPI_TL_IN_USE               107   /* Source endpoint in use already */
#define SNMPAPI_TL_TIMEOUT              108   /* No response within Timeout interval */
#define SNMPAPI_TL_TOO_BIG              109   /* PDU too big for send/receive */
#define SNMPAPI_TL_OTHER                199   /* Undefined transport error */
```

Specific transport layer errors are listed as appropriate in the definitions of the **SnmpRegister**, **SnmpSendMsg**, and **SnmpRecvMsg** functions later in this document.

Implementations should attempt to map specific transport errors to one of the generic transport errors. If no such mapping is possible, the implementation should return SNMPAPI_TL_OTHER. This error is preferred over SNMPAPI_OTHER_ERROR, for un-mapped transport layer errors..

2.11. WinSNMP Data Types

The following is an excerpt from the "Declarations" section of this document (and is part of the standard WinSNMP.h include file):

```
/* WinSNMP API Type Definitions  */
typedef  HANDLE                 HSNMP_SESSION,      FAR *LPHSNMP_SESSION;
typedef  HANDLE                 HSNMP_ENTITY,       FAR *LPHSNMP_ENTITY;
typedef  HANDLE                 HSNMP_CONTEXT,      FAR *LPHSNMP_CONTEXT;
typedef  HANDLE                 HSNMP_PDU,          FAR *LPHSNMP_PDU;
typedef  HANDLE                 HSNMP_VBL,          FAR *LPHSNMP_VBL;
typedef unsigned char           smiBYTE,            FAR *smiLPBYTE;
/* SNMP-related types from RFC1442 (SMI) */
typedef signed long             smiINT,             FAR *smiLPINT;
typedef smiINT                  smiINT32,           FAR *smiLPINT32;
typedef unsigned long           smiUINT32,          FAR *smiLPUINT32;
typedef struct {
    smiUINT32 len;
    smiLPBYTE ptr;}             smiOCTETS,          FAR *smiLPOCTETS;
typedef const smiOCTETS                             FAR *smiLPCOCTETS;
typedef smiOCTETS               smiBITS,            FAR *smiLPBITS;
typedef struct {
    smiUINT32   len;
    smiLPUINT32 ptr;}           smiOID,             FAR *smiLPOID;
typedef const smiOID                                FAR *smiLPCOID;
typedef smiOCTETS               smiIPADDR,          FAR *smiLPIPADDR;
typedef smiUINT32               smiCNTR32,          FAR *smiLPCNTR32;
typedef smiUINT32               smiGAUGE32,         FAR *smiLPGAUGE32;
typedef smiUINT32               smiTIMETICKS,       FAR *smiLPTIMETICKS;
typedef smiOCTETS               smiOPAQUE,          FAR *smiLPOPAQUE;
typedef smiOCTETS               smiNSAPADDR,        FAR *smiLPNSAPADDR;
typedef struct {
      smiUINT32 hipart;
      smiUINT32 lopart;}        smiCNTR64,          FAR *smiLPCNTR64;

/* Structure used to compose a value member for a variable binding */
typedef struct {                                    /* smiVALUE portion of VarBind */
        smiUINT32       syntax;                     /* Insert SNMP_SYNTAX_<type> */
        union {
        smiINT          sNumber;                    /* SNMP_SYNTAX_INT
                                                       SNMP_SYNTAX_INT32 */
        smiUINT32       uNumber;                    /* SNMP_SYNTAX_UINT32
                                                       SNMP_SYNTAX_CNTR32
                                                       SNMP_SYNTAX_GAUGE32
                                                       SNMP_SYNTAX_TIMETICKS */
        smiCNTR64       hNumber;                    /* SNMP_SYNTAX_CNTR64 */
        smiOCTETS       string;                     /* SNMP_SYNTAX_OCTETS
                                                       SNMP_SYNTAX_BITS
                                                       SNMP_SYNTAX_OPAQUE
                                                       SNMP_SYNTAX_IPADDR
                                                       SNMP_SYNTAX_NSAPADDR */
        smiOID          oid;                        /* SNMP_SYNTAX_OID */
        smiBYTE         empty;                      /* SNMP_SYNTAX_NULL
                                                       SNMP_SYNTAX_NOSUCHOBJECT
                                                       SNMP_SYNTAX_NOSUCHINSTANCE
                                                       SNMP_SYNTAX_ENDOFMIBVIEW */
              }         value;                      /* union */
        }               smiVALUE,                   FAR *smiLPVALUE;
typedef const smiVALUE                              FAR *smiLPCVALUE;
```

2.11.1. Integers

The "standard" integer type used in this specification is "unsigned long" (smiUINT32). In a few places, parameters are specified as "signed long" (smiINT) to comply with data elements defined in the respective RFCs. (This is especially true of some of the PDU components.)

2.11.2. Pointers

All pointer variables used in this specification are "far" pointers; large model programming is assumed.

2.11.3. Function Returns

All return values from WinSNMP functions fall into two categories:

- A HANDLE to a resource allocated by the implementation on behalf of the application, including:
    - Sessions (HSNMP_SESSION)
    - Entities (HSNMP_ENTITY)
    - Contexts (HSNMP_CONTEXT)
    - PDUs (HSNMP_PDU)
    - Variable Binding Lists (HSNMP_VBL)
- A long unsigned integer (smiUINT32) value representing a status (SNMPAPI_STATUS).
    - SNMPAPI_FAILURE (equates to 0 or NULL)
    - SNMPAPI_SUCCESS (equates to 1 or a positive count)

2.11.4. Descriptors

See Section 2.6.3. Descriptors, in Section 2.6. Memory Management.

Two important WinSNMP data types--namely, Octet Strings and Object Identifiers--take the form of "descriptors". A descriptor is a structure consisting of a length member ("len") and a pointer member (“ptr”), of the appropriate type (i.e., smiLPBYTE or smiLPUINT32, respectively), to the actual data item of interest. Note that either of these two descriptor types can occur in the “value” member of an smiVALUE structure, as can any of the “scalar” WinSNMP types.

When a descriptor which has been allocated by the application is actually populated (i.e., has its “len” and “ptr” members defined for it) by the implementation, the application must eventually call the **SnmpFreeDescriptor** function to enable the implementation to release the resources associated with “ptr” member.

3. WINDOWS SNMP INTERFACES

This section comprises the function reference for WinSNMP. In general, not a lot of significance attaches to the categorization or ordering of the functions. Some may argue that the "Entity/Context Functions" belong in the "Local Database Functions" category, or that the "Variable Binding Functions" belong in the "PDU Functions" category. Those, and similar assertions, could be true. The point for now is simply not to attach any significance to the grouping or the order of appearance.

- Local Database Functions
- Communications Functions
- Entity/Context Functions
- PDU Functions
- Variable Binding Functions
- Utility Functions

3.1. Local Database Functions

The functions in this section concern manipulation of the "local database" of SNMP administrative information.

The term "database" in this context is not meant to imply any particular data storage, access, or manipulation techniques. The WinSNMP implementation is the "owner" of the "local database" and may utilize any proprietary mechanisms it considers best, as long as all the functions defined in this section are fully supported and no additional implementation-specific functions are required of a WinSNMP application to utilize the "local database". Compliant WinSNMP implementations may require additional implementation-specific mechanisms external to a WinSNMP application (e.g., setting an environment variable in AUTOEXEC.BAT to point to a "local database" file or adjusting settings in a private <app>.ini file).

The functions in this section are:

Return Type	Procedure Name	Parameters
SNMPAPI_STATUS	SnmpGetTranslateMode	(OUT smiLPUINT32 nTranslateMode);
SNMPAPI_STATUS	SnmpSetTranslateMode	(IN smiUINT32 nTranslateMode);
SNMPAPI_STATUS	SnmpGetRetransmitMode	(OUT smiLPUINT32 nRetransmitMode);
SNMPAPI_STATUS	SnmpSetRetransmitMode	(IN smiUINT32 nRetransmitMode);
SNMPAPI_STATUS	SnmpGetTimeout	(IN HSNMP_ENTITY hEntity, OUT smiLPTIMETICKS nPolicyTimeout, OUT smiLPTIMETICKS nActualTimeout);
SNMPAPI_STATUS	SnmpSetTimeout	(IN HSNMP_ENTITY hEntity, IN smiTIMETICKS nPolicyTimeout);
SNMPAPI_STATUS	SnmpGetRetry	(IN HSNMP_ENTITY hEntity, OUT smiLPUINT32 nPolicyRetry, OUT smiLPUINT32 nActualRetry);
SNMPAPI_STATUS	SnmpSetRetry	(IN HSNMP_ENTITY hEntity, IN smiUINT32 nPolicyRetry);

3.1.1. SnmpGetTranslateMode()

The **SnmpGetTranslateMode** function informs the calling application as to the entity/context translation mode in effect at the time of the call..

Syntax:

```
SNMPAPI_STATUS      SnmpGetTranslateMode (
      OUT smiLPUINT32 nTranslateMode);
```

Parameter	Description
nTranslateMode	Pointer to variable to receive the current translation mode.

Returns:

The **SnmpGetTranslateMode** function returns SNMPAPI_SUCCESS if successful. In this case, the resultant value of nTranslateMode will be one of the following:

```
SNMPAPI_TRANSLATED          = Translate via Local Database look-up
SNMPAPI_UNTRANSLATED_V1 = Literal transport address and community string
SNMPAPI_UNTRANSLATED_V2 = Literal SNMPv2 party and context IDs
```

The **SnmpGetTranslateMode** function returns SNMPAPI_FAILURE if it fails. In this case, the value of nTranslateMode is undefined and meaningless to the application, and the value of **SnmpGetLastError** will be set to one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.

Comments:

See Section 2.3. Entity/Context Translation Modes.

3.1.2. SnmpSetTranslateMode()

The **SnmpSetTranslateMode** function enables the calling application to inform the implementation as the desired entity/context translation mode to use for subsequent **SnmpStrToEntity** and **SnmpStrToContext** function calls...

Syntax:

```
SNMPAPI_STATUS    SnmpSetTranslateMode (
    IN smiUINT32  nTranslateMode);
```

Parameter	Description
nTranslateMode	Value used to set the current translation mode--must be one of the following: SNMPAPI_TRANSLATED SNMPAPI_UNTRANSLATED_V1 SNMPAPI_UNTRANSLATED_V2

Returns:

The **SnmpSetTranslateMode** function returns SNMPAPI_SUCCESS if successful.

The **SnmpSetTranslateMode** function returns SNMPAPI_FAILURE if it fails and the value of **SnmpGetLastError** will be set to one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_MODE_INVALID	Indicates that the implementation does not support the requested translation mode.

Comments:

See Section 2.3. Entity/Context Translation Modes.

```
SNMPAPI_TRANSLATED          = Translate via Local Database look-up
SNMPAPI_UNTRANSLATED_V1 = Literal transport address and community string
SNMPAPI_UNTRANSLATED_V2 = Literal SNMPv2 party and context IDs
```

Upon successful execution of the **SnmpSetTranslateMode** function, the requested translation mode remains in effect for all subsequent **SnmpStrToEntity** and **SnmpStrToContext** function calls until another **SnmpSetTranslateMode** call with a different nTranslateMode value is executed successfully.

3.1.3. SnmpGetRetransmitMode()

The **SnmpGetRetransmitMode** function informs the calling application as to the retransmission mode in effect at the time of the call..

Syntax:

SNMPAPI_STATUS SnmpGetRetransmitMode (
OUT smiLPUINT32 nRetransmitMode);

Parameter	Description
nRetransmitMode	Pointer to variable to receive the current retransmission mode.

Returns:

The **SnmpGetRetransmitMode** function returns SNMPAPI_SUCCESS if successful. In this case, the resultant value of nRetransmitMode will be one of the following:

SNMPAPI_ON = The implementation ***is*** doing retransmission.
SNMPAPI_OFF = The implementation is ***not*** doing retransmission.

The **SnmpGetRetransmitMode** function returns SNMPAPI_FAILURE if it fails. In this case, the value of nRetransmitMode is undefined and meaningless to the application, and the value of **SnmpGetLastError** will be set to one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes

Comments:

See Section 2.8. Polling and Retransmission.

3.1.4. SnmpSetRetransmitMode()

The **SnmpSetRetransmitMode** function enables the calling application to inform the implementation as to the desired retransmission mode (i.e., timeout/retry) for subsequent **SnmpSendMsg** operations.

Syntax:

```
SNMPAPI_STATUS     SnmpSetRetransmitMode (
    IN smiUINT32   nRetransmitMode);
```

Parameter	Description
nRetransmitMode	Value used to set the current retransmission mode--must be one of the following: SNMPAPI_ON SNMPAPI_OFF

Returns:

The **SnmpSetRetransmitMode** function returns SNMPAPI_SUCCESS if successful.

The **SnmpSetRetransmitMode** function returns SNMPAPI_FAILURE if it fails and the value of **SnmpGetLastError** will be set to one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_MODE_INVALID	Indicates that the implementation does not support the requested translation mode.

Comments:

SNMPAPI_ON = The implementation ***is*** doing retransmission.
SNMPAPI_OFF = The implementation is ***not*** doing retransmission.

Changing the retransmission mode from SNMPAPI_OFF to SNMPAPI_ON has no effect on any SNMP communications initiated via **SnmpSendMsg** function calls which might be outstanding prior to successful return from the subject **SnmpSetRetransmitMode** function call. That is, an implementation does ***not*** have to execute the retransmission policy for messages which it initially sent when the retransmission mode was set to SNMPAPI_OFF and to which it has not yet received a response. An implementation ***may elect*** to execute the retransmission policy on behalf of such messages in this case, but this behavior is ***not a requirement*** and applications should not count on it. The purpose of this particular specification is to enable the implementations to take maximum advantage of the SNMPAPI_OFF retransmission mode when it is in effect.

When an application changes the retransmission mode from SNMPAPI_ON to SNMPAPI_OFF, the implementation ***should (but is not required to) cance***l all further retransmission attempts for any outstanding SNMP communications operations in effect prior to the call (and, of course, ***must not initiate*** any for subsequent **SnmpSendMsg** functions until the application might set the mode back to SNMPAPI_ON). Applications, however, should assume that the implementation has done so. The reason this behavior is so specified is that it might not be possible for an implementation run through a list of outstanding SNMP communications operations and turn each one off, while also receiving new **SnmpSendMsg** requests and traps and notifications from prior **SnmpRegister** requests, without one or more previously set retransmit timers waking up. Since this may be the "critical loop" for WinSNMP implementations, we need to ensure that the implementations can handle it efficiently.

3.1.5. SnmpGetTimeout()

The **SnmpGetTimeout** function returns current values for the retransmission timeout value on a per-entity basis. The timeout value is expressed in units of hundredths of seconds. The nPolicyTimeout value refers to the timeout value currently stored in the local database for the subject agent. The nActualTimeout value refers to the last measured or estimated response receipt interval reported by the implementation.

Syntax:

```
SNMPAPI_STATUS     SnmpGetTimeout (
    IN HSNMP_ENTITY        hEntity,
    OUT smiLPTIMETICKS     nPolicyTimeout,
    OUT smiLPTIMETICKS     nActualTimeout);
```

Parameter	Description
hEntity	Indicates the destination entity of interest.
nPolicyTimeout	Points to a variable to receive the timeout value for this entity as stored in the implementation's local database.
nActualTimeout	Points to a variable to receive the last measured or estimated response time interval from the destination agent.

Returns:

The **SnmpGetTimeout** function returns SNMPAPI_SUCCESS if successful.

The **SnmpGetTimeout** function returns SNMPAPI_FAILURE if it fails and the value of **SnmpGetLastError** will be set to one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_ENTITY_INVALID	Indicates that an entity parameter is invalid.

Comments:

See Section 2.8. Polling and Retransmission.

Implementations may provide utilities to load initial timeout values for the retransmission policy on a per destination entity basis, or may automatically assign some initial default value. Subsequent modifications to this value are made by applications with the **SnmpSetTimeout** function.

Implementations may or may not return measured or estimated values for the "actual timeout" parameter to the **SnmpGetTimeout** function. In the latter case, the implementation should return zero.

Applications should monitor the "actual timeout" value...if it is near, equal to, or greater than then current "policy timeout" value, the latter should be increased accordingly (or other corrective action taken).

3.1.6. SnmpSetTimeout()

The **SnmpSetTimeout** function enables an application to set the "policy timeout" value--in units of hundredths of seconds--on a per destination entity basis in the implementation's local database.

Syntax:

```
SNMPAPI_STATUS      SnmpSetTimeout (
      IN HSNMP_ENTITY hEntity,
      IN smiTIMETICKS  nPolicyTimeout);
```

Parameter	Description
hEntity	Indicates the destination entity of interest.
nPolicyTimeout	Indicates the timeout value for this entity to be stored in the implementation's local database.

Returns:

The **SnmpSetTimeout** function returns SNMPAPI_SUCCESS if successful.

The **SnmpSetTimeout** function returns SNMPAPI_FAILURE if it fails and the value of **SnmpGetLastError** will be set to one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_ENTITY_INVALID	Indicates that an entity parameter is invalid.

Comments:

See Section 2.8. Polling and Retransmission.

The timeout value is expressed in units of hundredths of seconds. If this value is zero, and both the application and the implementation agree to **SnmpSetRetransmitMode** (SNMPAPI_ON), then the implementation will select an operating value for this parameter when actually executing the retransmission policy.

3.1.7. SnmpGetRetry()

The **SnmpGetRetry** function returns current values for the retransmission retry value on a per-entity basis. The retry value is expressed as a unit count. The nPolicyRetry value refers to the retry value currently stored in the local database for the subject agent. The nActualRetry value refers to the last measured or estimated response retry count reported by the implementation.

Syntax:

```
SNMPAPI_STATUS      SnmpGetRetry (
      IN HSNMP_ENTITY  hEntity,
      OUT smiLPUINT32  nPolicyRetry,
      OUT smiLPUINT32   nActualRetry);
```

Parameter	Description
hEntity	Indicates the destination entity of interest.
nPolicyRetry	Points to a variable to receive the retry count value for this entity as stored in the implementation's local database.
nActualRetry	Points to a variable to receive the last measured or estimated response retry count from the destination agent.

Returns:

The **SnmpGetRetry** function returns SNMPAPI_SUCCESS if successful.

The **SnmpGetRetry** function returns SNMPAPI_FAILURE if it fails and the value of **SnmpGetLastError** will be set to one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_ENTITY_INVALID	Indicates that an entity parameter is invalid.

Comments:

See Section 2.8. Polling and Retransmission.

Implementations may provide utilities to load initial retry count values for the retransmission policy on a per destination entity basis, or may automatically assign some initial default value. Subsequent modifications to this value are made by applications with the **SnmpSetRetry** function.

Implementations may or may not return measured or estimated values for the "actual retry" parameter to the **SnmpGetRetry** function. In the latter case, the implementation should return zero.

Applications should monitor the "actual retry" value...if it is near, equal to, or greater than then current "policy retry" value, the latter should be increased accordingly (or other corrective action taken).

3.1.8. SnmpSetRetry()

The **SnmpSetRetry** function enables an application to set the "policy retry" count on a per destination entity basis in the implementation's local database.

Syntax:

```
SNMPAPI_STATUS      SnmpSetRetry (
      IN HSNMP_ENTITY hEntity,
      IN smiUINT32        nPolicyRetry);
```

Parameter	Description
hEntity	Indicates the destination entity of interest.
nPolicyRetry	Indicates the retry count for this entity to be stored in the implementation's local database.

Returns:

The **SnmpSetRetry** function returns SNMPAPI_SUCCESS if successful.

The **SnmpSetRetry** function returns SNMPAPI_FAILURE if it fails and the value of **SnmpGetLastError** will be set to one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_ENTITY_INVALID	Indicates that an entity parameter is invalid.

Comments:

See Section 2.8. Polling and Retransmission.

The retry value is expressed as a simple unit count. If this value is zero, and the application and the implementation have agreed to **SnmpSetRetransmitMode** (SNMPAPI_ON), then the implementation will select an operating value for this parameter when actually executing the retransmission policy.

3.2. Communications Functions

The functions in this section concern communications between the calling WinSNMP application and the serving WinSNMP implementation. Communications to and from other management entities--whether they reside on the local machine, on a connected LAN or WAN, or an internet--are handled by the WinSNMP implementation on behalf of the WinSNMP application, and without any overt orchestration by the latter.

The functions in this section are:

Return Type	Procedure Name	Parameters
SNMPAPI_STATUS	SnmpStartup	(OUT smiLPUINT32 nMajorVersion, OUT smiLPUINT32 nMinorVersion, OUT smiLPUINT32 nLevel, OUT smiLPUINT32 nTranslateMode, OUT smiLPUINT32 nRetransmitMode);
SNMPAPI_STATUS	SnmpCleanup	(void);
HSNMP_SESSION	SnmpOpen	(IN HWND hWnd, IN UINT wMsg);
SNMPAPI_STATUS	SnmpClose	(IN HSNMP_SESSION session);
SNMPAPI_STATUS	SnmpSendMsg	(IN HSNMP_SESSION session, IN HSNMP_ENTITY srcEntity, IN HSNMP_ENTITY dstEntity, IN HSNMP_CONTEXT context, IN HSNMP_PDU pdu);
SNMPAPI_STATUS	SnmpRecvMsg	(IN HSNMP_SESSION session, OUT LPHSNMP_ENTITY srcEntity, OUT LPHSNMP_ENTITY dstEntity, OUT LPHSNMP_CONTEXT context OUT LPHSNMP_PDU pdu);
SNMPAPI_STATUS	SnmpRegister	(IN HSNMP_SESSION session, IN HSNMP_ENTITY srcEntity, IN HSNMP_ENTITY dstEntity, IN HSNMP_CONTEXT context, IN smiLPCOID notification, IN smiUINT32 state);

3.2.1. SnmpStartup()

The **SnmpStartup** function notifies the implementation that the calling application is going to use its services, enabling the implementation to perform any required start-up procedures and allocations and to return some useful housekeeping information to the application.

Syntax:

```
SNMPAPI_STATUS     SnmpStartup (
      OUT smiLPUINT32        nMajorVersion,
      OUT smiLPUINT32        nMinorVersion,
      OUT smiLPUINT32        nLevel,
      OUT smiLPUINT32        nTranslateMode
      OUT smiLPUINT32        nRetransmitMode);
```

Parameter	Description
nMajorVersion	Pointer to variable to receive the major version number of the WinSNMP API implemented.
nMinorVersion	Pointer to variable to receive the minor version number of the WinSNMP API implemented.
nLevel	Pointer to variable to receive the highest level of SNMP communications supported by the implementation.
nTranslateMode	Pointer to variable to receive the default entity/context translation mode in effect for the implementation.
nRetransmitMode	Pointer to variable to receive the default retransmission mode in effect for the implementation.

Returns:

Upon success, the return value will be SNMPAPI_SUCCESS. In this case, the output parameters will contain appropriate values, as follows:

nMajorVersion will contain the major version number of the WinSNMP API implemented--the only legal value at this time is 1 (v1.nMinorVersion).

nMinorVersion will contain the minor version number of the WinSNMP API implemented--legal values at this time are 0 (v1.0) and 1 (v1.1).

nLevel will contain the highest level of SNMP communications supported by the implementation. This value may be one of the following:

```
SNMPAPI_NO_SUPPORT    = "Level 0" ( Message builder)
SNMPAPI_V1_SUPPORT    = "Level 1" (SNMPv1 agents)
SNMPAPI_V2_SUPPORT    = "Level 2" (SNMPv2 agents)
SNMPAPI_M2M_SUPPORT = "Level 3" (Manager-to-Manager)
```

nTranslateMode will contain the current default mode of translation of the entity and context parameters when used as inputs to **SnmpStrToEntity** and **SnmpStrToContext** functions. This value may be one of the following:

```
SNMPAPI_TRANSLATED          = Friendly names for translation via the Local Database
SNMPAPI_UNTRANSLATED_V1 = Literal SNMPv1 transport address and community string
SNMPAPI_UNTRANSLATED_V2 = Literal SNMPv2 partyID and contextID
```

nRetransmitMode will contain the current default retransmission mode in effect for the implementation. This value may be one of the following:

SNMPAPI_OFF= The implementation is ***not*** executing the retransmission policy
SNMPAPI_ON = The implementation ***is*** executing the retransmission policy

If this call fails, it will return SNMPAPI_FAILURE, and the application must use **SnmpGetLastError** to determine the reason.

SnmpGetLastError()	**Description**
"Common Error Codes"	See Section 2.10.1. Common Error Codes.

Comments:

Note: Every WinSNMP application must call **SnmpStartup** at least once and this call must precede any other WinSNMP API function call.

See Section 2.1. Levels of SNMP Support.

When this call fails, the application must not make any further WinSNMP API calls, other than **SnmpGetLastError** and, if appropriate, retries to **SnmpStartup**. If an application calls other WinSNMP API functions without a preceding successful **SnmpStartup**, the implementation should, if possible, return SNMPAPI_NOT_INITIALIZED.

An application which receives SNMPAPI_FAILURE and SNMP_ALLOC_ERROR in response to **SnmpStartup** may elect to wait or do other tasks and try again later in the hope that the implementation will have adequate free resources.

SnmpStartup is idempotent. This means that an application can call it multiple times with impunity. Multiple **SnmpStartup** calls do not require multiple **SnmpCleanup** calls. Every application must call **SnmpStartup** at least once, before any other WinSNMP API call, and must call **SnmpCleanup** at least once, as the last WinSNMP API call.

3.2.2. SnmpCleanup()

The **SnmpCleanup** function informs the implementation that the calling application is disconnecting and no longer requires any open resources which might be allocated to it by the implementation. The implementation will deallocate all resources allocated to the application, unless they have also been allocated to other active applications.

Syntax:

SNMPAPI_STATUS SnmpCleanup (void);

Returns:

The **SnmpCleanup** function returns SNMPAPI_SUCCESS if successful. Every subsequent WinSNMP API function call--until another successful **SnmpStartup** call--will return SNMPAPI_FAILURE with **SnmpGetLastError** set to report SNMPAPI_NOT_INITIALIZED.

The **SnmpCleanup** function returns SNMPAPI_FAILURE if it fails. The application should behave as though it had returned SNMPAPI_SUCCESS. As an additional step the application could call **SnmpGetLastError** to ascertain the reason for failure:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.

Comments:

Note: It is the responsibility of an application to use the respective **SnmpFree<xxx>** functions to free specific resources created an its behalf and to use **SnmpClose** to clean-up after every session opened via **SnmpOpen**. However, in the event that an application must perform an emergency exit and call **SnmpCleanup** without performing those steps, an implementation must perform all necessary clean-up of any resources under its control which were created on behalf of or otherwise allocated to that application. Even in this emergency situation, however, the application ***must*** call **SnmpCleanup** to enable this functionality in the implementation.

3.2.3. SnmpOpen()

The **SnmpOpen** function enables the implementation to allocate and initialize memory, resources, and/or communications mechanisms and data structures for the application. The application will continue to use the "session identifier" returned by the implementation in subsequent WinSNMP function calls to facilitate resource accounting on a per session basis. This mechanism will enable the implementation to perform an orderly release of resources in response to a subsequent **SnmpClose** function call for a given session.

Syntax:

```
HSNMP_SESSION     SnmpOpen (
    IN HWND       hWnd,
    IN UINT       wMsg);
```

Parameter	Description
hWnd	Identifies the application's notification window.
wMsg	Identifies the application's notification message.

Returns:

If the function is successful, the return value is a HANDLE which identifies the WinSNMP session opened by the implementation on behalf of the calling application.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_HWND_INVALID	The hWnd parameter is not a valid window handle.

Comments:

See Section 2.5. Sessions.

An application can open multiple sessions. Each such session for the same hWnd should provide a different wMsg, but this is not required. A successful **SnmpOpen** call using always returns a unique session handle (with respect to all other currently open sessions for the calling application).

The hWnd parameter specifies the window handle to be notified when an asynchronous request completes or trap/notification occurs and the wMsg parameter specifies the message number that the window will be sent. Upon receipt of this message, the application should call **SnmpRecvMsg** to retrieve the subject PDU for immediate or subsequent processing.

In other programming models (e.g., synchronous, CLI-driven, or "curtained" applications), the WinSNMP implementation may interpret hWnd and hMsg differently. Likewise, the session model ***may*** be used to facilitate multi-threaded programming in supporting environments.

Note: A well-behaved WinSNMP application will call **SnmpClose** for each session opened by **SnmpOpen**. When an emergency exit is required of the application, it ***must*** at least call **SnmpCleanup**. A well-behaved WinSNMP implementation ***must*** react to an **SnmpCleanup** call as though it were a series of **SnmpClose** calls for each open session allocated to the calling application.

3.2.4. SnmpClose()

The **SnmpClose** function causes the implementation to deallocate and/or close memory, resources, communications mechanisms and data structures associated with the specified session, on behalf of the calling application.

Syntax:

```
SNMPAPI_STATUS      SnmpClose (
      IN HSNMP_SESSION session);
```

Parameter	Description
session	A handle specifying the session to close.

Returns:

SNMPAPI_SUCCESS if the function successfully closes the WinSNMP session

If **SnmpClose** fails, it will return SNMPAPI_FAILURE. Use **SnmpGetLastError** to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates session parameter is invalid.

Comments:

Closing a session on which asynchronous requests are outstanding will cause any outstanding requests and/or replies for that session to be discarded by the implementation.

Note: A well-behaved WinSNMP application will call **SnmpClose** for each session opened by **SnmpOpen**. When an emergency exit is required of the application, it ***must*** at least call **SnmpCleanup**. A well-behaved WinSNMP implementation ***must*** react to an **SnmpCleanup** call as though it were a series of **SnmpClose** calls for each open session allocated to the calling application.

3.2.5. SnmpSendMsg()

The **SnmpSendMsg** function requests the specified PDU be transmitted to the destination entity, using the specified context and--for SNMPv2 communications--the designated source entity

When a transmission request is received by the implementation via the **SnmpSendMsg** function, the WinSNMP implementation determines which version of SNMP and which transport to use based on its own capabilities and the corresponding properties associated with the requesting session and with the remote entity which holds the context to be accessed, based on values in the Local Database.

Syntax:

```
SNMPAPI_STATUS      SnmpSendMsg (
      IN  HSNMP_SESSION      session,
      IN HSNMP_ENTITY        srcEntity,
      IN HSNMP_ENTITY        dstEntity,
      IN HSNMP_CONTEXT       context,
      IN  HSNMP_PDU          pdu);
```

Parameter	Description
session	Identifies the session that will perform the operation.
srcEntity	Identifies the subject management entity.
dstEntity	Identifies the target management entity.
context	Identifies the target context of interest.
pdu	Identifies the SNMP protocol data unit containing the operation.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use **SnmpGetLastError** to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates that a session parameter is invalid.
SNMPAPI_ENTITY_INVALID	Indicates that an entity parameter is invalid.
SNMPAPI_CONTEXT_INVALID	Indicates that the context parameter is invalid.
SNMPAPI_PDU_INVALID	Indicates that the PDU parameter is invalid.
SNMPAPI_OPERATION_INVALID	Indicates that the PDU_type element is inappropriate for the destination entity.
SNMPAPI_TL_NOT_INITIALIZED	Transport layer not initialized
SNMPAPI_TL_NOT_SUPPORTED	Transport does not support protocol
SNMPAPI_TL_NOT_AVAILABLE	Network subsystem has failed
SNMPAPI_TL_RESOURCE_ERROR	Transport resource error
SNMPAPI_TL_SRC_INVALID	Source endpoint invalid
SNMPAPI_TL_INVALID_PARAM	Invalid parameter to transport call
SNMPAPI_TL_PDU_TOO_BIG	PDU was too big for transport
SNMPAPI_TL_OTHER	An undefined transport error occurred

Comments:

See Section 2.2. Transport Interface Support, Section 2.7. Asynchronous Model, Section 2.8. Polling and Retransmission, and Section 2.9. RequestIDs for related background information.

This function returns immediately. If the return indicates an error, **SnmpGetLastError** should be called immediately to find out the error type. When the asynchronous request completes, the hWnd specified in the **SnmpOpen** call is sent the wMsg specified. The application should call **SnmpRecvMsg** with this HSNMP_SESSION to retrieve the results from the request.

Note: It is the responsibility of the WinSNMP implementation to verify the correctness of the PDU structure (and other arguments) and to return failure to the caller and an extended error code via **SnmpGetLastError**. For example, for a PDU_type other than SNMP_PDU_GETBULK and SNMP_PDU_RESPONSE (if allowed), passed values (other than zero) for error_status and/or error_index would constitute an invalid PDU structure and the implementation should return SNMPAPI_FAILURE and set **SnmpGetLastError** to report SNMPAPI_PDU_INVALID.

Note: An application may assign a RequestID to a PDU at any time via the **SnmpCreatePdu** or **SnmpSetPduData** functions. If the RequestID component is zero at the time of the **SnmpCreatePdu** call, the implementation will assign a non-zero RequestID to the PDU. An application that wants to use a zero-valued RequestID must set it to that value with the **SnmpSetPduData** function.

Note: As SNMP replies do not necessarily come back in the same order as requests were sent, the application should check the RequestID of the received message to match it with the appropriate request.

If an SNMPv2 feature is requested, but the dstEntity implies an entity using SNMPv1, then the down-grading procedures defined in the SNMPv2 "coexistence" specification (RFC1452) are used. If down-grading is not possible (e.g., an InformRequest-PDU directed at an SNMPv1 agent), then the function will fail and **SnmpGetLastError** will return SNMPAPI_OPERATION_INVALID.

3.2.6. SnmpRecvMsg()

The **SnmpRecvMsg** function retrieves the results from a completed asynchronous request made on a given HSNMP_SESSION. It also receives traps registered for that session.

Syntax:

```
SNMPAPI_STATUS     SnmpRecvMsg (
      IN  HSNMP_SESSION          session,
      OUT LPHSNMP_ENTITY         srcEntity,
      OUT LPHSNMP_ENTITY         dstEntity,
      OUT LPHSNMP_CONTEXT        context
      OUT LPHSNMP_PDU            pdu);
```

Parameter	Description
session	Specifies the session that will receive the SNMP message.
srcEntity	Identifies the entity (agent role) which sent the message.
dstEntity	Identifies the entity (manager role) which is to receive the message.
context	Identifies the context from which the srcEntity issued the message.
pdu	Identifies the PDU component of the received message.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS, and the OUT parameters are populated with their corresponding values.

If the function fails, the return value is SNMPAPI_FAILURE. Note that for the Transport Layer (TL) errors, the OUT parameters are populated with their corresponding values to enable applications to recover gracefully. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates that the session parameter is invalid.
SNMPAPI_NOOP	Indicates that this session has no messages in its queue at this time
SNMPAPI_TL_NOT_INITIALIZED	Transport layer not initialized
SNMPAPI_TL_NOT_SUPPORTED	Transport does not support protocol
SNMPAPI_TL_NOT_AVAILABLE	Network subsystem has failed
SNMPAPI_TL_RESOURCE_ERROR	Transport resource error
SNMPAPI_TL_UNDELIVERABLE	Destination unreachable
SNMPAPI_TL_SRC_INVALID	Source endpoint invalid
SNMPAPI_TL_INVALID_PARAM	Invalid parameter to transport call
SNMPAPI_TL_PDU_TOO_BIG	PDU was too big for transport
SNMPAPI_TL_TIMEOUT	No response within Timeout Interval
SNMPAPI_TL_OTHER	Undefined transport error

Comments:

See Section 2.2. Transport Interface Support, Section 2.7. Asynchronous Model, Section 2.8. Polling and Retransmission, and Section 2.9. RequestIDs for related background information.

The implementation is only required to deliver information via **SnmpRecvMsg** that it has access to in the SNMP message it received from the transport layer. For SNMPv2, all components are included in the SNMP message itself. For SNMPv1, an implementation has several choices: It might have access to additional transport layer data and elect to use that; it can probably associate an in-bound GetResponse PDU with an out-bound request PDU and use the srcEntity and dstEntity values from that; or it can return NULL for components missing from the received SNMP message.

The application is responsible for freeing the HANDLE object resources returned by this function when it is no longer needed by the application, by calling the **SnmpFreePdu**, **SnmpFreeEntity**, and **SnmpFreeContext** functions when appropriate.

Note that there are four HANDLE objects instantiated by a successful **SnmpRecvMsg** operation (i.e., the varbindlist component of the returned PDU is **not** instantiated until called for by the application via the **SnmpGetPduData** function.

Replies are not necessarily received in the same order as their originating requests were sent. For traps received from SNMPv1 entities, in addition to mapping them to SNMPv2 format, the implementation must assign a non-zero RequestID. A RequestID value delivered via trap notification can possibly duplicate a RequestID used by an application on a request PDU; applications need to check for this occurrence.

When a trap is delivered by **SnmpRecvMsg**, it is returned in the SNMPv2 format, even if a SNMPv1 entity generated the trap. The SNMPv2 "coexistence" specification, as described in RFC 1452, specifies the mapping rules between the SNMPv1 and SNMPv2 trap formats. However, for the convenience of management applications, the final variable binding for a SNMPv1-generated trap will always be snmpTrapEnterpriseOID.0, even if the trap is a generic trap such as coldStart. See Appendix A. Mapping Traps Between SNMPv1 and SNMPv2.

3.2.7. SnmpRegister()

The **SnmpRegister** function registers the application's desire to receive or discontinue trap and inform notifications from the specified entity of interest (dstEntity), which will act in an agent role.

Syntax:

```
SNMPAPI_STATUS      SnmpRegister (
      IN HSNMP_SESSION      session,
      IN HSNMP_ENTITY       srcEntity,
      IN HSNMP_ENTITY       dstEntity,
      IN HSNMP_CONTEXT      context,
      IN smiLPCOID          notification,
      IN smiUINT32          status);
```

Parameter	Description
session	Identifies the session which is interested in registering.
srcEntity	Identifies the management entity (manager role) of interest--this will be the trap recipient. (This is the “source” of the notification request.)
dstEntity	Identifies the management entity (agent role) of interest--this will be the trap sender. (This is the “target” of the notification request.)
context	Identifies the context of interest.
notification	Identifies the trap/notification OID matching sequence to be registered or un-registered.
status	Indicates whether to register (SNMPAPI_ON) or un-register (SNMPAPI_OFF) for the subject notification..

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes..
SNMPAPI_SESSION_INVALID	Indicates that the session parameter is invalid.
SNMPAPI_ENTITY_INVALID	Indicates that the entity parameter is invalid.
SNMPAPI_CONTEXT_INVALID	Indicates that the context parameter is invalid.
SNMPAPI_OID_INVALID	Indicates that the notification parameter is invalid.
SNMPAPI_TL_NOT_INITIALIZED	Transport layer not initialized
SNMPAPI_TL_IN_USE	Trap port not available
SNMPAPI_TL_NOT_AVAILABLE	Network subsystem has failed

Comments:

In WinSNMP all traps are delivered to the applications are SNMPv2 traps. If an implementation receives an SNMPv1 trap from an SNMPv1 agent, it must convert it to an SNMPv2 trap in accordance with RFC 1452 (the “Coexistence” document).

See Appendix A. Mapping Traps Between SNMPv1 and SNMPv2.

Notifications, traps or informs, are defined using OBJECT IDENTIFIERs, as specified in SNMPv2. Hence, an application interested in receiving coldStart traps should construct an OBJECT IDENTIFIER corresponding to this trap based upon the SNMPv2 MIB (RFC 1450) and use this as the notification parameter.

The value of the notification parameter is used for pattern matching against the OIDs of received traps and notifications. That is, if the first ‘n’ sub-ids of a received SnmpTrapOID match all the sub-ids (‘n’) of a notification value passed to **SnmpRegister**, than that SnmpTrapOID is a match. Accordingly, a received SnmpTrapOID with fewer sub-ids than a given notification parameters must fail the matching process with respect to that particular notification parameter.

An application may pass NULL for any or all of the srcEntity, dstEntity, context, and notification parameters. The significance of NULL in any of these parameters is, effectively, to tell the implementation to ***not*** filter out any received traps or notifications on the basis of this parameter.

If the **notification** parameter is NULL, then the application is indicating that it is interested in registering or unregistering for any and all notifications from the dstEntity, as indicated by the **status** parameter.

If the **status** parameter contains any value other than SNMPAPI_OFF or SNMPAPI_ON, it will be treated as though it were SNMPAPI_ON.

Upon receipt of a trap/notification, the hWnd parameter specified in the **SnmpOpen** call for the registered **session** is sent the wMsg specified. The application should call **SnmpRecvMsg** with this **session** to retrieve the appropriate results.

Note that it is the responsibility of a Level 3 implementation to acknowledge the receipt of an InformRequest-PDU. This tells the issuing management entity that the inform made it to the implementation "platform", but not necessarily to any particular application(s).

In the case where a NULL dstEntity parameter to **SnmpRegister** results in the implementation creating an entity object for the srcEntity parameter on a future **SnmpRecvMsg** call, the entity will "belong" to the application as though it had caused its creation with **SnmpStrToEntity**. Put differently, the behavior in this respect will be the same as for **SnmpDecodeMsg**. This is equally true--although perhaps less likely to occur--with respect to the srcEntity and context parameters as well.

Note that this functionality relates to [not] filtering traps/notifications ***received*** by the implementation. It does not address the issue of how such traps/notifications get directed to the implementation in the first place. This is assumed to occur "out-of-band" from the perspective of application making use of NULL filtering parameters as described above.

3.3. Entity/Context Functions

The functions in this section enable the application to use human-oriented string identifiers for the entity and context "objects" and concepts, while permitting the WinSNMP implementation to adopt proprietary repository, access method, and runtime representation strategies vis-à-vis the "local database", entities, parties, and contexts.

The functions in this section are:

Return Type	Procedure Name	Parameters
HSNMP_ENTITY	SnmpStrToEntity	(IN HSNMP_SESSION session, IN LPCSTR entity);
SNMPAPI_STATUS	SnmpEntityToStr	(IN HSNMP_ENTITY entity, IN smiUINT32 size, OUT LPSTR string);
SNMPAPI_STATUS	SnmpFreeEntity	(IN HSNMP_ENTITY entity);
HSNMP_CONTEXT	SnmpStrToContext	(IN HSNMP_SESSION session, IN smiLPCOCTETS string);
SNMPAPI_STATUS	SnmpContextToStr	(IN HSNMP_CONTEXT context, OUT smiLPOCTETS string);
SNMPAPI_STATUS	SnmpFreeContext	(IN HSNMP_CONTEXT context);

3.3.1. SnmpStrToEntity()

The **SnmpStrToEntity** function accepts a pointer to a null-terminated text string identifying an entity of interest and, if successful, returns a handle to an implementation-specific representation of entity information. Note that the resulting entity handle may be used as either a srcEntity value or as a dstEntity value. Note, also, that the semantics of the input string are governed by the value of entity/context translation mode in effect at the time of the call.

Syntax:

```
HSNMP_ENTITY        SnmpStrToEntity (
    IN HSNMP_SESSION        session,
    IN LPCSTR               entity);
```

Parameter	Description
session	Handle of the allocating session.
entity	Pointer to a NULL-terminated text string identifying the management entity of interest.

Returns:

If the function is successful, the return value is an HSNMP_ENTITY handle.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes..
SNMPAPI_SESSION_INVALID	Indicates an invalid session handle.
SNMPAPI_ENTITY_UNKNOWN	Indicates entity parameter is unknown.

Comments:

See Section 2.3. Entity/Context Translation Modes.

When the application no longer needs to utilize this entity handle, the **SnmpFreeEntity** function should be called to release the resources associated with it.

When the current entity/context translation mode is SNMPAPI_TRANSLATED, the "entity" parameter is assumed to be a user-friendly textual name to be de-referenced via the Local Database.

When the current entity/context translation mode is SNMPAPI_UNTRANSLATED_V1, the "entity" parameter is assumed to be a literal transport address (in textual form). The implementation will attempt to identify Local Database resources associated with this SNMPv1 "address" and will supply working defaults when no such entry exists in the Local Database. This is to enable "out-of-the-box" SNMPv1/UDP operation with WinSNMP.

When the current entity/context translation mode is SNMPAPI_UNTRANSLATED_V2, the "entity" parameter is assumed to be a literal PartyID (in textual form). The implementation will attempt to identify Local Database resources associated with this SNMPv2 "party" and will supply working defaults when no such entry exists in the Local Database. This is to enable "out-of-the-box" SNMPv2/ InitialPartyID operation with WinSNMP.

3.3.2. SnmpEntityToStr()

The **SnmpEntityToStr** function returns a string value identifying an entity.

Syntax:

```
SNMPAPI_STATUS      SnmpEntityToStr (
    IN HSNMP_ENTITY         entity,
    IN smiUINT32            size,
    OUT LPSTR               string);
```

Parameter	Description
entity	A handle specifying an entity.
size	The size of the buffer the application is providing to contain the string.
string	Points to a buffer that will receive the NULL-terminated string that identifies the management entity.

Returns:

If the function is successful, the return value is the number of bytes, including the NULL terminating byte, output to "string"--this value may be less than or equal to "size", but not greater.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_ENTITY_INVALID	Indicates entity parameter is unknown.
SNMPAPI_OUTPUT_TRUNCATED	Indicates that the buffer was too small.

Comments:

See Section 2.3. Entity/Context Translation Modes.

Note that the current setting of the entity/context translation mode affects this function:

If the setting is SNMPAPI_TRANSLATED, then the implementation returns the user-friendly textual name of this entity from the Local Database. If no such name exists in the Local Database, then the function returns either of the following, depending upon whether the entity is known to be SNMPv1 or SNMPv2.

If the setting is SNMPAPI_UNTRANSLATED_V1 and the subject entity is an SNMPv1 creature, then the implementations returns the transport address of the entity (in textual form). If the subject entity is an SNMPv2 creature, then the implementation behaves as though the entity/context translation mode setting were SNMPAPI_UNTRANSLATED_V2 for the purposes of this call only.

If the setting is SNMPAPI_UNTRANSLATED_V2 and the subject entity is an SNMPv2 creature, then the implementations returns the PartyID of the entity (in textual form). If the subject entity is an SNMPv1 creature, then the implementation behaves as though the entity/context translation mode setting were SNMPAPI_UNTRANSLATED_V1 for the purposes of this call only.

3.3.3. SnmpFreeEntity()

The **SnmpFreeEntity** function releases resources associated with a entity returned by the **SnmpStrToEntity** function.

Syntax:

```
SNMPAPI_STATUS      SnmpFreeEntity (
      IN HSNMP_ENTITY         entity);
```

Parameter	Description
entity	An entity handle to be released.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_ENTITY_INVALID	Indicates entity parameter is invalid.

Comments:

Un-freed resources created on behalf of the application will be freed by the implementation upon execution of an associated **SnmpClose** function or upon execution of an **SnmpCleanup** function. Nonetheless, a well-behaved WinSNMP application will individually free all such resources using the atomic "free" functions. The reason for this is to eliminate or, at least, minimize any "batch-like" loads on the implementation, so that other applications can be serviced in a timely fashion.

3.3.4. SnmpStrToContext()

The **SnmpStrToContext** function accepts an OCTET STRING naming the collection of managed objects (or "profile") of interest (for SNMPAPI_TRANSLATED mode), a community string (for SNMPAPI_UNTRANSLATED_V1 mode), or a contextID (for SNMPAPI_UNTRANSLATED_V2 mode) and returns a handle to an implementation-specific representation of context information for use with the **SnmpSendMsg** and **SnmpRegister** functions.

Syntax:

```
HSNMP_CONTEXT     SnmpStrToContext (
    IN HSNMP_SESSION      session,
    IN smiLPCOCTETS       string);
```

Parameter	Description
session	Handle of the allocating session.
string	Pointer to an smiOCTETS descriptor identifying a collection of managed objects, community string, or contextID.

Returns:

If the function is successful, the return value is an HSNMP_CONTEXT handle.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes..
SNMPAPI_SESSION_INVALID	Indicates an invalid session handle.
SNMPAPI_CONTEXT_INVALID	Indicates that the string descriptor is invalid (e.g., len and/or ptr member is NULL).
SNMPAPI_CONTEXT_UNKNOWN	Indicates that the value referenced in the string descriptor is unknown.

Comments:

See Section 2.3. Entity/Context Translation Modes.

Note: The smiOCTETS descriptor used for the **string** parameter in the **SnmpStrToContext** function is both allocated and populated by the application. Hence, **SnmpFreeDescriptor** should **not** be called to free the memory associated with the **ptr** member of this descriptor.

Note: "Strings" referenced in descriptors (such as an smiOCTETS structure) do not require a NULL terminating byte. Such a string can be used in an IN smiOCTETS parameter by merely setting the **len** member to ignore it.

When the application no longer needs to utilize this context handle, the **SnmpFreeContext** function should be called to release the resources associated with it.

Note that the current setting of the entity/context translation mode affects this function:

When the current entity/context translation mode is SNMPAPI_TRANSLATED, the "string" parameter is assumed to describe a user-friendly name (in textual form) to be de-referenced via the Local Database.

When the current entity/context translation mode is SNMPAPI_UNTRANSLATED_V1, the "string" parameter is assumed to describe a literal community string (which may contain non-printable ASCII byte values).

When the current entity/context translation mode is SNMPAPI_UNTRANSLATED_V2, the "string" parameter is assumed to describe a literal ContextID (in textual form). The implementation will attempt to identify Local Database resources associated with this SNMPv2 "context" and will supply working defaults when no such entry exists in the Local Database. This is to enable "out-of-the-box" SNMPv2/ InitialContextID operation with WinSNMP.

3.3.5. SnmpContextToStr()

The **SnmpContextToStr** function populates an smiOCTETS descriptor with a context value appropriate to the entity/context translation mode in effect at the time of execution..

Syntax:

```
SNMPAPI_STATUS      SnmpContextToStr (
      IN HSNMP_CONTEXT      context,
      OUT smiLPOCTETS       string);
```

Parameter	Description
context	A handle specifying a context.
string	A pointer to an smiOCTETS descriptor buffer that will receive the string which identifies the context.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_CONTEXT_INVALID	Indicates that the context handle is invalid.

Comments:

See Section 2.3. Entity/Context Translation Modes.

Note: The application provides only the address of a valid smiOCTETS descriptor structure as the **string** parameter. The implementation, upon successful execution of the **SnmpContextToStr** function, will populate the **len** and **ptr** members of the descriptor. The application must call **SnmpFreeDescriptor** --when appropriate--to enable the implementation to free the memory resources so consumed.

Note: “Strings” referenced in descriptors (such as an smiOCTETS structure) do not require a NULL terminating byte. Applications should not expect a NULL-terminated string to be returned in an OUTput smiOCTETS parameter.

Note that the current setting of the entity/context translation mode affects this function:

If the setting is SNMPAPI_TRANSLATED, then the implementation returns the user-friendly textual name of this **context** from the Local Database. If no such name exists in the Local Database, then the function returns either of the following, depending upon whether the **context** is known to be an SNMPv1 or SNMPv2 construct.

If the setting is SNMPAPI_UNTRANSLATED_V1 and the subject **context** is an SNMPv1 construct, then the implementation returns the raw community string (which may contain non-printable byte values). If the subject **context** is an SNMPv2 construct, then the implementation behaves as though the entity/context translation mode setting were SNMPAPI_UNTRANSLATED_V2 for the purposes of this call only.

If the setting is SNMPAPI_UNTRANSLATED_V2 and the subject **context** is an SNMPv2 construct, then the implementations returns the raw ContextID (in textual form). If the subject entity is an SNMPv1 construct, then the implementation behaves as though the entity/context translation mode setting were SNMPAPI_UNTRANSLATED_V1 for the purposes of this call only.

3.3.6. SnmpFreeContext()

The **SnmpFreeContext** function releases resources associated with a context returned by the **SnmpStrToContext** function.

Syntax:

```
SNMPAPI_STATUS     SnmpFreeContext (
    IN HSNMP_CONTEXT      context);
```

Parameter	Description
context	A context handle to be released.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_CONTEXT_INVALID	Indicates context parameter is invalid.

Comments:

Un-freed resources created on behalf of the application will be freed by the implementation upon execution of an associated **SnmpClose** function or upon execution of an **SnmpCleanup** function. Nonetheless, a well-behaved WinSNMP application will individually free all such resources using the atomic "free" functions. The reason for this is to eliminate or, at least, minimize any "batch-like" loads on the implementation, so that other applications can be serviced in a timely fashion.

3.4. PDU Functions

This section defines functions which construct PDUs for use in the **SnmpSendMsg** and **SnmpEncodeMsg** functions and which decompose PDUs received via the **SnmpRecvMsg** and **SnmpDecodeMsg** functions. The following section--Variable Binding Functions--also pertains to PDU [de]composition, but is retained as a separate section both for consistency with the earlier versions of this document and for modularization.

Actual PDU and variable binding data structures are private to the WinSNMP implementation. The PDU and Variable Binding functions enable applications to extract the component data elements which are then available for whatever use the application deems appropriate. The elements comprising a PDU from the perspective of a WinSNMP application are:

```
/* This typedef is for expository purposes only.  It is not a required component of WinSNMP */
typedef struct {
        smiINT          PDU_type;
        smiINT32        request_id;
        smiINT          error_status;     -- "non_repeaters" for BulkPDU
        smiINT          error_index;      -- "max_repetitions" for BulkPDU
        HSNMP_VBL       varbindlist;}     -- we'll examine this one in the next section
                        PDU;
```

The functions in this section are:

Return Type	Procedure Name	Parameters
HSNMP_PDU	SnmpCreatePdu	(IN HSNMP_SESSION session, IN smiINT PDU_type, IN smiINT32 request_id, IN smiINT error_status/non_repeaters, IN smiINT error_index/max_repetitions, IN HSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpGetPduData	(IN HSNMP_PDU PDU, OUT smiLPINT PDU_type, OUT smiLPINT32 request_id, OUT smiLPINT error_status/non_repeaters, OUT smiLPINT error_index/max_repetitions, OUT LPHSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpSetPduData	(IN HSNMP_PDU PDU, IN const smiINT FAR *PDU_type, IN const smiINT32 FAR *request_id, IN const smiINT FAR *non_repeaters, IN const smiINT FAR *max_repetitions, IN const HSNMP_VBL FAR *vbl);
HSNMP_PDU	SnmpDuplicatePdu	(IN HSNMP_SESSION session, IN HSNMP_PDU PDU);
SNMPAPI_STATUS	SnmpFreePdu	(IN HSNMP_PDU PDU);

The following table illustrates the possible PDU_type values used in WinSNMP functions:

PDU_types Table

SNMP_PDU_GET	Indicates a Get Request-PDU
SNMP_PDU_GETNEXT	Indicates a GetNextRequest-PDU
SNMP_PDU_GETBULK	Indicates a GetBulkRequest-PDU
SNMP_PDU_V1TRAP	Indicates an SNMPv1-Trap-PDU
SNMP_PDU_SET	Indicates a SetRequest- PDU
SNMP_PDU_INFORM	Indicates an InformRequest-PDU
SNMP_PDU_RESPONSE	Indicates a Response-PDU
SNMP_PDU_TRAP	Indicates an SNMPv2-Trap-PDU

The following table illustrates the possible SNMP error values used in the error_status element of an SNMP PDU:

SNMP Error Values Table

SNMP_ERROR_NOERROR	Specifies the noError error.
SNMP_ERROR_TOOBIG	Specifies the tooBig error.
SNMP_ERROR_NOSUCHNAME	Specifies the noSuchName error.
SNMP_ERROR_BADVALUE	Specifies the badValue error.
SNMP_ERROR_READONLY	Specifies the readOnly error.
SNMP_ERROR_GENERR	Specifies the genErr error.
SNMP_ERROR_NOACCESS	Specifies the noAccess error.
SNMP_ERROR_WRONGTYPE	Specifies the wrongType error.
SNMP_ERROR_WRONGLENGTH	Specifies the wrongLength error.
SNMP_ERROR_WRONGENCODING	Specifies the wrongEncoding error.
SNMP_ERROR_WRONGVALUE	Specifies the wrongValue error.
SNMP_ERROR_NOCREATION	Specifies the noCreation error.
SNMP_ERROR_INCONSISTENTVALUE	Specifies the inconsistentValue error.
SNMP_ERROR_RESOURCEUNAVAILABLE	Specifies the resourceUnavailable error.
SNMP_ERROR_COMMITFAILED	Specifies the commitFailed error.
SNMP_ERROR_UNDOFAILED	Specifies the undoFailed error.
SNMP_ERROR_AUTHORIZATIONERROR	Specifies the authorizationError error.
SNMP_ERROR_NOTWRITABLE	Specifies the notWritable error.
SNMP_ERROR_INCONSISTENTNAME	Specifies the inconsistentName error.

3.4.1. SnmpCreatePdu()

The **SnmpCreatePdu** function allocates and initializes an SNMP protocol data unit for subsequent use in **SnmpSendMsg**, **SnmpEncodeMsg**, and other functions.

Note that all input parameters to **SnmpCreatePdu** must be present; but, with the exception of the "session" parameter, all may be NULL, resulting in a default PDU as defined below.

Syntax:

```
HSNMP_PDU           SnmpCreatePdu (
    IN HSNMP_SESSION    session,
    IN smiINT           PDU_type,
    IN smiINT32         request_id,
    IN smiINT           error_status,    -- "non_repeaters" for BulkPDU
    IN smiINT           error_index,     -- "max_repetitions" for BulkPDU
    IN HSNMP_VBL        vbl;
```

Parameter	Description
session	Handle of the allocating session.
PDU_type	NULL or one of the values shown in the PDU_types table shown in the introduction to the PDU Functions section. If NULL, the WinSNMP implementation will supply SNMP_PDU_GETNEXT.
request_id	An application-supplied value used to identify the PDU or NULL, in which case the WinSNMP implementation will supply a value.
error_status	Ignored (and may be NULL) on input for all PDU types except SNMP_PDU_GETBULK, in which case it represents the value for non_repeaters. For all other PDU types, the WinSNMP implementation will supply SNMP_ERROR_NOERROR.
error_index	Ignored (and may be NULL) on input for all PDU types except SNMP_PDU_GETBULK, in which case it represents the value for max_repetitions. The WinSNMP implementation returns 0 (zero) for all other PDU types.
vbl	A handle to a varbindlist data structure (or NULL).

Returns:

If the function is successful, the return value identifies the created SNMP protocol data unit

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates an invalid session handle.
SNMPAPI_PDU_INVALID	Indicates an invalid PDU_type value.
SNMPAPI_VBL_INVALID	Indicates an invalid vbl.

Comments:

Assuming NULL values for all input parameters (other than "session"), the created protocol data unit defaults to the following attributes:

PDU_type:	SNMP_PDU_GETNEXT
request_id:	<WinSNMP-generated value>
error_status:	SNMP_ERROR_NOERROR
error_index:	0
vbl:	NULL

After completing operations with the created PDU, the **SnmpFreePdu** function should be called to release the resources allocated to the PDU by the **SnmpCreatePdu** function..

3.4.2. SnmpGetPduData()

The **SnmpGetPduData** function extracts selected data elements from the specified PDU and copies them to the respective locations given as corresponding output parameters.

Note that all output parameters must be supplied to the function call, but any (or all) of them may be NULL. No values are returned for output parameters passed as NULL.

Syntax:

```
SNMPAPI_STATUS          SnmpGetPduData (
    IN HSNMP_PDU        PDU,
    OUT smiLPINT        PDU_type,
    OUT smiLPINT32      request_id,
    OUT smiLPINT        error_status,    -- "non_repeaters" for GetBulkRequest-PDU
    OUT smiLPINT        error_index,     -- "max_repetitions" for GetBulkRequest-PDU
    OUT LPHSNMP_VBL     vbl);
```

Parameter	Description
PDU	Identifies the SNMP protocol data unit.
PDU_type	If not NULL, points to an smiINT variable that will receive the PDU_type of the PDU.
request_id	If not NULL, points to an smiINT32 variable that will receive the request_id of the PDU.
error_status	If not NULL, points to an smiINT variable that will receive the error_status (or non_repeaters) of the PDU.
error_index	If not NULL, points to an smiINT variable that will receive the error_index (or max_repetitions) of the PDU.
vbl	If not NULL, points to an HSNMP_VBL variable that will receive the handle to the varbindlist of the PDU.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes..
SNMPAPI_PDU_INVALID	Indicates that the PDU parameter is invalid.
SNMPAPI_NOOP	Indicates that all output parameters were NULL

Comments:

On a successful return and if the parameter was not NULL, PDU_type will contain one of the values from the PDU_Types Table shown in the introduction to the PDU Functions section.

On a successful return and if the parameter was not NULL, error_status will contain one of the values from the SNMP Error Values Table shown in the introduction to the PDU Functions section.

As always, a well-behaved application must handle the case when an unexpected value (PDU_type and error_status are just possible examples) might be returned by a procedure call.

It is very important to note that the HSNMP_VBL resource returned if the call is successful and the vbl parameter was not NULL represents a ***new*** varbindlist object. This is consistent both with the primary

purpose of the **SnmpGetPduData** function--namely, to dereference the elements of a ***newly received*** PDU and with the WinSNMP policy that says that anytime the implementation returns a handled resource object to the application, it will be a newly allocated object. For PDUs that an application has either created for sending or has already dereferenced with **SnmpGetPduData**, the application is expected to "know" the values of the PDU components. Calling **SnmpGetPduData** with a non-NULL vbl parameter against an existing dereferenced PDU is equivalent (wrt the returned HSNMP_VBL object) to calling **SnmpDuplicateVbl** on the vbl object already attached to the PDU.

Calling **SnmpGetPduData** with a non-NULL vbl parameter against a ***newly received*** PDU has the effect of "attaching" the returned HSNMP_VBL object to the HSNMP_PDU object specified in the input pdu parameter. Calling **SnmpGetPduData** with a non-NULL vbl parameter against an existing (created or dereferenced) PDU, results in a new HSNMP_VBL object as discussed above but does not disturb the original HSNMP_VBL object value already attached to the PDU.

3.4.3. SnmpSetPduData()

The **SnmpSetPduData** function updates selected data elements in the specified PDU.

Note that all parameters must be supplied to the function call, but any (or all) of them--except the PDU--may be NULL. No values are changed in the PDU for input parameters passed as NULL (and they are passed as pointers to values to allow for the case when NULL is the desired update **value**).

Syntax:

```
SNMPAPI_STATUS      SnmpSetPduData (
    IN HSNMP_PDU    PDU,
    IN smiLPINT     PDU_type,
    IN smiLPINT32   request_id,
    IN smiLPINT     non_repeaters,      -- for GetBulkRequest-PDU only
    IN smiLPINT     max_repetitions,    -- for GetBulkRequest-PDU only
    IN LPHSNMP_VBL  vbl);
```

Parameter	Description
PDU	Identifies the SNMP protocol data unit.
PDU_type	If not NULL, points to an smiINT variable that will update the PDU_type of the PDU.
request_id	If not NULL, points to an smiINT32 variable that will update the request_id of the PDU.
non_repeaters	If not NULL, points to an smiINT variable that will update the non_repeaters of the GetBulkRequest-PDU (ignored for other PDU_types).
max_repetitions	If not NULL, points to an smiINT variable that will update the max_repetitions of the GetBulkRequest-PDU (ignored for other PDU_types).
vbl	If not NULL, points to an HSNMP_VBL variable that will update the handle to the varbindlist of the PDU.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_PDU_INVALID	Indicates that the PDU parameter is invalid.
SNMPAPI_VBL_INVALID	Indicates an invalid vbl parameter.
SNMPAPI_NOOP	Indicates that all input parameters were NULL

Comments:

Not all possible combinations of individually legal component values are valid. Ultimately, the WinSNMP implementation must verify the validity of the PDU (and other message elements) when the application calls the **SnmpSendMsg** or **SnmpEncodeMsg** function and reject and ill-formed or otherwise illegal PDU structures

3.4.4. SnmpDuplicatePdu()

The **SnmpDuplicatePdu** function duplicates an SNMP protocol data unit structure identified by the PDU parameter.

Syntax:

```
HSNMP_PDU               SnmpDuplicatePdu (
    IN HSNMP_SESSION    session,
    IN HSNMP_PDU        PDU);
```

Parameter	Description
session	Handle of the allocating session.
PDU	Identifies the SNMP protocol data unit to duplicate.

Returns:

If the function is successful, the return value is a handle which identifies the new (duplicated) SNMP protocol data unit.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates an invalid session handle.
SNMPAPI_PDU_INVALID	Indicates that the PDU parameter is invalid.

Comments:

After using the duplicated message, **SnmpFreePdu** function should be called to release the resources allocated to the PDU by the **SnmpDuplicatePdu** function.

Note: The handle returned by a successful call to **SnmpDuplicatePdu** will be unique among active PDU handles, at least within the calling application.

3.4.5. SnmpFreePdu()

The **SnmpFreePdu** function releases resources associated with a protocol data unit previously created by the **SnmpCreatePdu** or **SnmpDuplicatePdu** function.

Syntax:

```
SNMPAPI_STATUS      SnmpFreePdu (
    IN HSNMP_PDU    PDU);
```

Parameter	Description
PDU	Identifies the SNMP protocol data unit to be freed.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_PDU_INVALID	Indicates that the PDU parameter is invalid.

Comments:

Un-freed resources created on behalf of the application will be freed by the implementation upon execution of an associated **SnmpClose** function or upon execution of an **SnmpCleanup** function. Nonetheless, a well-behaved WinSNMP application will individually free all such resources using the atomic "free" functions. The reason for this is to eliminate or, at least, minimize any "batch-like" loads on the implementation, so that other applications can be serviced in a timely fashion.

Varbinds and VarBindLists are re-usable independently of any given PDU. In WinSNMP, a varbind does not exist outside of a varbindlist (even if the latter consists of only a single varbind). There is a separate atomic function--**SnmpFreeVbl**--to deallocate varbindlist resources. Of course, upon execution of **SnmpFreePdu**, the WinSNMP implementation must free any ***internal*** resources allocated to VBLs for that PDU. That's different from the HSNMP_VBL resources requested and held by a session in the calling application.

3.5. Variable Binding Functions

WinSNMP relies on a varbindlist structure (VBL), and drops the concept of a separate varbind structure (VB). No capability is lost, since an individual varbind structure can be represented by a varbindlist structure of one member. A WinSNMP **application** accesses the varbindlist structure via handles--of type HSNMP_VBL. A WinSNMP **implementation** hides the details of this structure from the application using whatever proprietary mechanisms and techniques it considers optimal.

These functions allow applications to easily construct and manipulate VarBindLists for inclusion in PDUs. Note that a varbind is not directly associated with a PDU, only indirectly through inclusion in a varbindlist. A varbindlist gets associated with and de-referenced from a PDU with the **SnmpSetPduData** and **SnmpGetPduData**, respectively.

The functions in this section are:

Return Type	Procedure Name	Parameters
HSNMP_VBL	SnmpCreateVbl	(IN HSNMP_SESSION session, IN smiLPCOID name, IN smiLPCVALUE value);
HSNMP_VBL	SnmpDuplicateVbl	(IN HSNMP_SESSION session, IN HSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpFreeVbl	(IN HSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpCountVbl	(IN HSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpGetVb	(IN HSNMP_VBL vbl, IN smiUINT32 index, OUT smiLPOID name, OUT smiLPVALUE value);
SNMPAPI_STATUS	SnmpSetVb	(IN HSNMP_VBL vbl, IN smiUINT32 index, IN smiLPCOID name, IN smiLPCVALUE value);
SNMPAPI_STATUS	SnmpDeleteVb	(IN HSNMP_VBL vbl, IN smiUINT32 index);

Table of Syntax Values Used in Variable Binding Data Structures

SNMP_SYNTAX_INT32
SNMP_SYNTAX_OCTETS
SNMP_SYNTAX_OID
SNMP_SYNTAX_BITS
SNMP_SYNTAX_IPADDR
SNMP_SYNTAX_CNTR32
SNMP_SYNTAX_GAUGE32
SNMP_SYNTAX_TIMETICKS
SNMP_SYNTAX_OPAQUE
SNMP_SYNTAX_NSAPADDR
SNMP_SYNTAX_CNTR64
SNMP_SYNTAX_UINT32
SNMP_SYNTAX_NULL
SNMP_SYNTAX_NOSUCHOBJECT
SNMP_SYNTAX_NOSUCHINSTANCE
SNMP_SYNTAX_ENDOFMIBVIEW

3.5.1. SnmpCreateVbl()

The **SnmpCreateVbl** function creates a new varbindlist structure for the calling application. If the "name" and "value" parameters are not NULL, **SnmpCreateVbl** uses them to construct the initial varbind member of the varbindlist.

Syntax:

```
HSNMP_VBL               SnmpCreateVbl (
    IN HSNMP_SESSION    session,
    IN smiLPCOID        name,
    IN smiLPCVALUE      value);
```

Parameter	Description
session	Handle of the allocating session.
name	If not NULL, points to an OID for initialization of the varbindlist.
value	If not NULL, points to a value for initialization of the varbidlist.

Returns:

If the function is successful, the return value identifies a newly created varbindlist structure.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates an invalid session handle.
SNMPAPI_OID_INVALID	Indicates that the name parameter referenced an invalid OID structure.
SNMPAPI_SYNTAX_INVALID	Indicates that the syntax field of the value parameter is invalid.

Comments:

IF the "name" parameter is not NULL and the "value" parameter is NULL, the varbindlist will still be initialized, with the value set to NULL and with syntax of SNMP_SYNTAX_NULL. If the "name" parameters is NULL, the varbindlist will not be initialized, and the "value" parameter will be ignored.

Every call to **SnmpCreateVbl** must be matched with a corresponding call to **SnmpFreeVbl** to release the resources associated with the varbindlist. A memory leak will result if a variable used to hold an HSNMP_VBL value returned by **SnmpCreateVbl** (or **SnmpDuplicateVbl**) is re-used for a subsequent **SnmpCreateVbl** (or **SnmpDuplicateVbl**) operation before it has been passed to **SnmpFreeVbl**.

3.5.2. SnmpDuplicateVbl()

The **SnmpDuplicateVbl** function creates a new varbindlist structure for the specified session in the calling application and initializes it with a copy of the input vbl (which may be empty).

Syntax:

```
HSNMP_VBL               SnmpDuplicateVbl (
    IN HSNMP_SESSION    session,
    IN HSNMP_VBL        vbl);
```

Parameter	Description
session	Handle of the allocating session.
vbl	Identifies the varbindlist to be duplicated.

Returns:

If the function is successful, the return value identifies a newly created varbindlist structure.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates an invalid session handle.
SNMPAPI_VBL_INVALID	Indicates that the vbl parameter is invalid.

Comments:

Every call to **SnmpDuplicateVbl** must be matched with a corresponding call to **SnmpFreeVbl** to release the resources associated with the varbindlist. A memory leak will result if a variable used to hold an HSNMP_VBL value returned by **SnmpDuplicateVbl** (or **SnmpCreateVbl**) is re-used for a subsequent **SnmpDuplicateVbl** (or **SnmpCreateVbl**) operation before it has been passed to **SnmpFreeVbl**.

Note: The handle returned by a successful call to **SnmpDuplicateVbl** will be unique among active VBL handles, at least within the calling application.

3.5.3. SnmpFreeVbl()

The **SnmpFreeVbl** function releases resources associated with a varbindlist structure previously allocated by **SnmpCreateVbl** or **SnmpDuplicateVbl**. It is the responsibility of WinSNMP applications to free varbindlist resources allocated through calls to **SnmpCreateVbl** and **SnmpDuplicateVbl**.

Syntax:

```
SNMPAPI_STATUS      SnmpFreeVbl (
    IN HSNMP_VBL     vbl);
```

Parameter	Description
vbl	Identifies the varbindlist to be released.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes..
SNMPAPI_VBL_INVALID	Indicates that the vbl parameter is invalid.

Comments:

Every call to **SnmpCreateVbl** must be matched with a corresponding call to **SnmpFreeVbl** to release the resources associated with the varbindlist. A memory leak will result if a variable used to hold an HSNMP_VBL value returned by **SnmpCreateVbl** is re-used for a subsequent **SnmpCreateVbl** operation before it has been passed to **SnmpFreeVbl**. The foregoing comments apply equally to VarBindLists originating via the **SnmpDuplicateVbl** function.

Un-freed resources created on behalf of the application will be freed by the implementation upon execution of an associated **SnmpClose** function or upon execution of an **SnmpCleanup** function. Nonetheless, a well-behaved WinSNMP application will individually free all such resources using the atomic "free" functions. The reason for this is to eliminate or, at least, minimize any "batch-like" loads on the implementation, so that other applications can be serviced in a timely fashion.

3.5.4. SnmpCountVbl()

The **SnmpCountVbl** function counts the number of varbinds in the varbindlist identified by the vbl input parameter.

Syntax:

```
SNMPAPI_STATUS      SnmpCountVbl (
IN HSNMP_VBL        vbl);
```

Parameter	Description
vbl	Identifies the subject varbindlist.

Returns:

If the function is successful, the return value is the count of varbinds in the varbindlist.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_VBL_INVALID	Indicates that the vbl parameter is invalid.
SNMPAPI_NOOP	Indicates that the vbl resources contained no varbinds at this time.

Comments:

The value returned when the result is SNMPAPI_SUCCESS represents the maximum "index" value in the **SnmpGetVb** and **SnmpSetVb** functions.

3.5.5. SnmpGetVb()

The **SnmpGetVb** function retrieves the object instance name and its associated value from the varbind identified by the index parameter. The **SnmpGetVb** function returns the object instance name in the descriptor pointed to by the name parameter and its associated value in the descriptor pointed to by the value parameter.

Syntax:

```
SNMPAPI_STATUS      SnmpGetVb (
      IN HSNMP_VBL          vbl,
      IN smiUINT32          index,
      OUT smiLPOID          name,
      OUT smiLPVALUE        value);
```

Parameter	Description
vbl	Identifies the subject varbindlist.
index	Identifies the position of the subject varbind within the varbindlist.
name	Points to a variable to receive the OID portion of the varbind.
value	Points to a variable to receive the value portion of the varbind.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_VBL_INVALID	Indicates that the vbl parameter is invalid.
SNMPAPI_INDEX_INVALID	Indicates that the index parameter is invalid.

Comments:

Valid values for the "index" parameter come from the **SnmpCountVbl** function and from the "error_index" component of GetResponse PDUs returned via the **SnmpRecvMsg** function. These values range from 1 to n, where n is the total number of varbinds in the varbindlist.

The member elements of the smiOID and smiVALUE structures pointed to by the “name” and “value” parameters are ignored on input and will be overwritten by the implementation upon a successful execution of this function.

On a successful return, the syntax field of the "value" variable will contain one of the object syntax types shown in the Table of Syntax Values included in the introduction to this section.

The application must eventually call the **SnmpFreeDescriptor** function to enable the implementation to free any resources that might have been allocated to populate the “ptr” members of the “name” and (depending upon its “syntax” member) “value” structures.

3.5.6. SnmpSetVb()

The **SnmpSetVb** function adds and updates varbind entries in a varbindlist.

Syntax:

```
SNMPAPI_STATUS      SnmpSetVb (
    IN HSNMP_VBL    vbl,
    IN smiUINT32    index,
    IN smiLPCOID    name,
    IN smiLPCVALUE  value);
```

Parameter	Description
vbl	Identifies the target varbindlist.
index	Identifies the position of the subject varbind within the varbindlist for an "update" operation or is zero for an "add/append" operation.
name	Points to a variable containing the object instance name to be set.
value	If not NULL, points to a variable containing the associated value to be set.

Returns:

If the function is successful, the return value is the position ("index" value) of the affected varbind. In the case of successful update operations, the return value will equal the "index" parameter. In the case of add/append operations (in which the "index" parameter will have been zero), the return value will be "n+1", where "n" was the previous total count of varbinds in the varbindlist (per **SnmpCountVbl**).

f the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_VBL_INVALID	Indicates that the vbl parameter is invalid.
SNMPAPI_INDEX_INVALID	Indicates that the index parameter is invalid.
SNMPAPI_OID_INVALID	Indicates that the name parameter is invalid.
SNMPAPI_SYNTAX_INVALID	Indicates that the syntax field of the value parameter is invalid.

Comments:

Valid values for the "index" parameter range from 0 (zero) to n, where n is the total number of varbinds currently in the varbindlist as reported by the **SnmpCountVbl** function. An index value of 0 (zero) indicates the addition of a varbind to the varbindlist.

If the "value" parameter is NULL, the varbind will still be initialized, with the value set to NULL and with syntax of SNMP_SYNTAX_NULL.

3.5.7. SnmpDeleteVb()

The SnmpDeleteVb function removes a varbind entry from a varbindlist.

Syntax:

```
SNMPAPI_STATUS     SnmpDeleteVb (
    IN HSNMP_VBL   vbl,
    IN smiUINT32   index);
```

Parameter	Description
vbl	Identifies the target varbindlist.
index	Identifies the position of the subject varbind within the varbindlist.

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_VBL_INVALID	Indicates that the vbl parameter is invalid.
SNMPAPI_INDEX_INVALID	Indicates that the index parameter is invalid.

Comments:

Valid values for the "index" parameter come from the **SnmpCountVbl** function and from the "error_index" component of GetResponse PDUs returned via the **SnmpRecvMsg** function. These values range from 1 to n, where n is the total number of varbinds in the varbindlist.

A typical use for this function will be when a GetResponse PDU includes an SNMP error and the user elects to resubmit the original "request" PDU *sans* the offending varbind.

Note that following a successful SnmpDeleteVb operation, any varbinds that previously came after the deleted varbind will logically “move up” in the varbindlist--that is, their index values will decrement by one position and the total number of varbinds in the varbindlist, as returned by SnmpCountVbl will likewise decrement by one.

It is legal to end up with an empty varbindlist by executing “SnmpDeleteVb (hVBL, 1)” on the last remaining varbind in a varbindlist. In this case, the varbindlist itself (as a HANDLE’d object) is still valid and must eventually be released via SnmpFreeVbl.

Sample pseudo-code for **SnmpDeleteVb**:

```
-- Omitting error-checking the function calls for clarity's sake...
nReqID = SnmpRecvMsg (session, &rSrc, &rDst, &rCtx, &rPDU);
SnmpGetPduData (rPDU, &rType, &rReqid, &rErrstat, &rErridx, &rVBL);
-- Assuming type == GetResponse-PDU and
-- Assuming error_status != SNMP_ERROR_NOERROR...
-- Assuming the error is something we cannot or do not want to fix...
        -- If error_index == 1, do an SnmpCountVbl ();
        -- if count <= 1 follow another strategy (like SnmpFreeVbl (sVBL))
-- Assuming error_index > 1 || count > 1...
SnmpDeleteVb (sVBL, rErridx);
-- And assuming we want to re-try the SNMP operation
-- ...with a new Request_ID just for good measure...
sReqid++;
SnmpSetPduData (sPDU, NULL, &sReqid, NULL, NULL, &sVBL);
SnmpSendMsg (session, sSrc, sDst, sCtx, sPDU);
-- No need for the received PDU or VBL any longer...
SnmpFreePdu (rPDU);
SnmpFreeVbl (rVBL);
-- Go back to doing what we were doing before all of this started...
```

3.6. Utility Functions

The utility functions are offered to ease the tasks of bookkeeping and dealing with objects passed across the Windows SNMP interface.

Return Type	Procedure Name	Parameters
SNMPAPI_STATUS	SnmpGetLastError	(IN HSNMP_SESSION session);
SNMPAPI_STATUS	SnmpStrToOid	(IN LPCSTR string, OUT smiLPOID dstOID);
SNMPAPI_STATUS	SnmpOidToStr	(IN smiLPCOID srcOID, IN smiUINT32 size, OUT LPSTR string);
SNMPAPI_STATUS	SnmpOidCopy	(IN smiLPCOID srcOID, OUT smiLPOID dstOID);
SNMPAPI_STATUS	SnmpOidCompare	(IN smiLPCOID xOID, IN smiLPCOID yOID, IN smiUINT32 maxlen, OUT smiLPINT result);
SNMPAPI_STATUS	SnmpEncodeMsg	(IN HSNMP_SESSION session, IN HSNMP_ENTITY srcEntity, IN HSNMP_ENTITY dstEntity, IN HSNMP_CONTEXT context, IN HSNMP_PDU pdu, OUT smiLPOCTETS msgBufDesc);
SNMPAPI_STATUS	SnmpDecodeMsg	(IN HSNMP_SESSION session, OUT LPHSNMP_ENTITY srcEntity, OUT LPHSNMP_ENTITY dstEntity, OUT LPHSNMP_CONTEXT context, OUT LPHSNMP_PDU pdu, IN smiLPCOCTETS msgBufDesc);
SNMPAPI_STATUS	SnmpFreeDescriptor	(IN smiUINT32 syntax, IN smiLPOPAQUE descriptor);

3.6.1. SnmpGetLastError()

The **SnmpGetLastError** function returns an indication of why the last WinSNMP operation executed by the application failed.

Syntax:

```
SNMPAPI_STATUS              SnmpGetLastError (
    IN HSNMP_SESSION        session);
```

Parameter	Description
session	Indicates the session for which error information is requested. If NULL, the application-wide error information is returned.

Returns:

This function returns the last WinSNMP error that occurred for the indicated session or for the application (task) if the session is NULL (as when, for example, **SnmpStartup** fails.)

Comments:

See Section 2.10. Error Handling.

This function should be called immediately after any API call that fails, as the value is overwritten after each API call which fails.

The session input parameter is provided to facilitate accommodation of multi-threaded Windows operating environments. Single-threaded applications can always pass a NULL session value and retrieve the last error information for the overall application.

Note that **SnmpGetLastError** must be able to return a value to a WinSNMP application even when **SnmpStartup** fails, and/or before any sessions are created with **SnmpOpen**, and/or after all sessions are closed with **SnmpClose** and/or the application disconnects from the implementation with the **SnmpCleanup** function.

3.6.2. SnmpStrToOid()

The **SnmpStrToOid** function converts a textual representation of the dotted numeric form of an OBJECT IDENTIFIER into an internal OBJECT IDENTIFIER representation.

Syntax:

```
SNMPAPI_STATUS    SnmpStrToOid (
    IN  LPCSTR        string,
    OUT smiLPOID      dstOID);
```

Parameter	Description
string	Points to a NULL terminated string to be converted.
dstOID	Points to an smiOID variable to receive the converted value.

Returns:

If the function is successful, the return value is the number of sub-identifiers in the output OBJECT IDENTIFIER. This number will also be the value of the "len" member of the dstOID structure upon return..

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_OID_INVALID	Indicates that the string was invalid.

Comments:

Note that the format of the OID array in an smiOID structure is one integral sub-identifier per array element. That is, the string “1.3.6.1” (lstrlen=7) becomes an array of {1,3,6,1} (len=4) and vice versa.

The member elements of the smiOID structure pointed to by the dstOID structure are ignored on input and will be overwritten by the implementation upon a successful execution of this function.

The application must eventually call the **SnmpFreeDescriptor** function to enable the implementation to free any resources that might have been allocated to populate the “ptr” member of the dstOID structure.

This function has nothing to do with the MIB database APIs described elsewhere. The purpose of this function is to translate from the dotted numeric string representation of an OID (e.g. "1.2.3.4.5.6") to the internal object identifier format.

This function can fail with SNMPAPI_OID_INVALID, for example, if the "string" input parameter is not NULL terminated, is of insufficient length, is longer than MAXOBJIDSTRSIZE, or does not constitute the textual form of a valid OID.

3.6.3. SnmpOidToStr()

The **SnmpOidToStr** function converts an internal representation of an OBJECT IDENTIFIER into a dotted numeric string representation of an OBJECT IDENTIFIER.

Syntax:

```
SNMPAPI_STATUS    SnmpOidToStr (
    IN  smiLPCOID     srcOID,
    IN smiUINT32      size,
    OUT LPSTR         string);
```

Parameter	Description
srcOID	Points to a variable holding an object identifier to be converted.
size	The size of the buffer the application is providing to contain the string.
string	Points to a buffer that will receive the string that identifies the management entity.

Returns:

If the function is successful, the return value is the number of bytes, including the NULL terminating byte, output to "string"--this value may be less than or equal to "size", but not greater.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes..
SNMPAPI_OID_INVALID	Indicates that the srcOID was invalid.
SNMPAPI_OUTPUT_TRUNCATED	Indicates that the buffer was too small.

Comments:

This function has nothing to do with the MIB database APIs described elsewhere. The purpose of this function is to translate from the internal object identifier format to the dotted numeric string representation of the OID (e.g., "1.2.3.4.5.6").

Note that the format of the OID array in an smiOID structure is one integral sub-identifier per array element. That is, the string “1.3.6.1” (lstrlen=7) becomes an array of {1,3,6,1} (len=4) and vice versa.

Note that the application should use a string buffer of MAXOBJIDSTRSIZE length for this call, to be safe. If, as will normally be true, a shorter OID is actually decoded, the application can copy the resulting string to one of appropriate length and either re-use or free the space allocated to the original buffer.

Note that a NULL-terminated string is returned for convenience. The return value, upon success, will include the terminating NULL byte.

3.6.4. SnmpOidCopy()

The **SnmpOidCopy** function copies the srcOID to the dstOID.

Syntax:

```
SNMPAPI_STATUS      SnmpOidCopy (
    IN smiLPCOID    srcOID,
    OUT smiLPOID    dstOID);
```

Parameter	Description
srcOID	Points to a variable holding an object identifier.
dstOID	Points to a variable to receive a copy of the srcOID.

Returns:

If the function is successful, the return value is the number of sub-identifiers in the output OBJECT IDENTIFIER. This number will also be the value of the "len" member of the dstOID structure upon return.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_OID_INVALID	Indicates that the srcOID was invalid.

Comments:

The member elements of the smiOID structure pointed to by the dstOID structure are ignored on input and will be overwritten by the implementation upon a successful execution of this function.

The application must eventually call the **SnmpFreeDescriptor** function to enable the implementation to free any resources that might have been allocated to populate the “ptr” member of the dstOID structure.

3.6.5. SnmpOidCompare()

The **SnmpOidCompare** function lexicographically compares two OIDs. If "maxlen" is non-zero, then its value is used as an upper limit on the number of sub-identifiers to compare. This approach will most often be used to identify whether two OIDs have common prefixes or not. If "maxlen" is zero, then the "len" members of the two smiOID structures will determine the maximum number of sub-identifiers to compare.

Syntax:

```
SNMPAPI_STATUS    SnmpOidCompare (
    IN smiLPCOID  xOID,
    IN smiLPCOID  yOID,
    IN smiUINT32  maxlen,
    OUT smiLPINT  result);
```

Parameter	Description
xOID	Points to a variable holding an object identifier to compare.
yOID	Points to a variable holding an object identifier to compare.
maxlen	If non-zero, Indicates the number of sub-identifiers to compare. Must be less than MAXOBJIDSIZE.
result	Points to a variable to receive the result of the comparison: > 0 if xOID is greater than yOID = 0 if xOID equals yOID < 0 if xOID is less than yOID

Returns:

If the function is successful, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. Use the **SnmpGetLastError** function to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_OID_INVALID	Indicates that either or both of the input OIDs were invalid.
SNMPAPI_SIZE_INVALID	Indicates that the "maxlen" parameter was invalid; that is, greater than MAXOBJIDSIZE.

Comments:

The **SnmpOidCompare** function combines the functionality of the **SnmpOidCmp** and **SnmpOidNCmp** functions which appeared in versions earlier that v1.0g of this specification.

When "maxlen" is non-zero (but not greater than MAXOBJIDSIZE), the maximum number of sub-identifiers that will be compared is the ***minimum*** of the "maxlen" input parameter and the two "len" members of the input OID structures. Either or both of the input OIDs can have a zero length without causing an error.

When "maxlen" is zero, the maximum number of sub-identifiers that will be compared is the minimum of the two "len" members of the input OID structures. Either or both of the input OIDs can have a zero length without causing an error.

If the two OIDs are lexicographically equal when the maximum number of sub-identifiers have been compared, then:

- If the "maxlen" parameter value was used as the maximum number of sub-identifiers to compare, or if the two OID parameters have equal "len" members which are less than the "maxlen" input parameter, the "result" value will be 0 (equal).

- If an OID "len" member was used as the value for the maximum number of sub-identifiers to compare (because it was less than the non-zero "maxlen" input parameter or because "maxlen" was equal to zero), and the other OID "len" member value is greater, the "result" value will be <0 or >0, depending on which OID parameter had which "len" value.

3.6.6. SnmpEncodeMsg()

The **SnmpEncodeMsg** routine takes as its first five input parameters the same parameters passed to SnmpSendMsg. The implementation will use these parameters to form an SNMP "message" as though they had arrived via the SnmpSendMsg function. The implementation will not, however, attempt to transmit the resulting message to the 'dstEntity'. It will, instead, use the 'msgBufDesc' parameter as described herein to return to the application the encoded/serialized SNMP message that it would have transmitted to the 'dstEntity' if SnmpSendMsg had been called.

Syntax:

```
SNMPAPI_STATUS      SnmpEncodeMsg (
      IN HSNMP_SESSION      session,
      IN HSNMP_ENTITY       srcEntity,
      IN HSNMP_ENTITY       dstEntity,
      IN HSNMP_CONTEXT      context,
      IN HSNMP_PDU          pdu,
      OUT smiLPOCTETS       msgBufDesc);
```

Parameter	Description
session	Identifies the session that will perform the operation.
srcEntity	Identifies the subject management entity.
dstEntity	Identifies the target management entity.
context	Identifies the target context of interest.
PDU	Identifies the SNMP protocol data unit containing the requested operation.
msgBufDesc	Identifies the variable to receive the encoded SNMP message.

Returns:

Upon success, **SnmpEncodeMsg** returns the length, in bytes, of the encoded SNMP message. (This value will also be in the “len” member of the msgBufDesc output parameter.)

If the function fails, the return value is SNMPAPI_FAILURE. Use **SnmpGetLastError** to obtain extended error information.

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates that a session parameter is invalid.
SNMPAPI_ENTITY_INVALID	Indicates that an entity parameter is invalid.
SNMPAPI_CONTEXT_INVALID	Indicates that the context parameter is invalid.
SNMPAPI_PDU_INVALID	Indicates that the PDU parameter is invalid.

Comments:

The member elements of the smiOCTETS structure pointed to by the msgBufDesc structure are ignored and will be overwritten by the implementation upon a successful execution of this function.

The application must eventually call the **SnmpFreeDescriptor** function to enable the implementation to free any resources that might have been allocated to populate the “ptr” member of the msgBufDesc structure.

If any of the first five input parameters fail the normal integrity checks performed for **SnmpSendMsg** then **SnmpEncodeMsg** will return SNMPAPI_FAILURE and **SnmpGetLastError** will be set to return the appropriate extended error code.

3.6.7. SnmpDecodeMsg()

The **SnmpDecodeMsg** function is the converse of the **SnmpEncodeMsg** function. It takes as input a session identifier and a far pointer to an smiOCTETS structure which describes an encoded/serialized SNMP message to be decoded into its constituent components. The session identifier is required since new resources will be created by the implementation and allocated to the application as a result of calling this function, if it is successful. The 'msgBufDesc' input parameter consists of two elements: The "len" member identifies the maximum number of bytes to process; the "ptr" member points to the encoded/serialized SNMP message to decode.

Syntax:

```
SNMPAPI_STATUS      SnmpDecodeMsg (
      IN HSNMP_SESSION              session,
      OUT LPHSNMP_ENTITY            srcEntity,
      OUT LPHSNMP_ENTITY            dstEntity,
      OUT LPHSNMP_CONTEXT           context,
      OUT LPHSNMP_PDU               pdu,
      IN smiLPCOCTETS               msgBufDesc);
```

Parameter	Description
session	Identifies the session that will perform the operation.
srcEntity	Identifies the subject management entity.
dstEntity	Identifies the target management entity.
context	Identifies the target context of interest.
PDU	Identifies the SNMP protocol data unit.
msgBufDesc	Identifies the buffer holding the encoded SNMP message.

Returns:

Upon successful completion, **SnmpDecodeMsg** returns the actual number of bytes decoded. This may be equal to or less than the "len" member of the 'msgBufDesc' input parameter. Also, upon success, **SnmpDecodeMsg** returns handle values in the 'srcEntity', 'dstEntity', 'context', and 'pdu' output parameters. Note that these resources are to be freed by the application using the appropriate **SnmpFree**<xxx> functions, or by the implementation in response to an **SnmpClose** or **SnmpCleanup** function call.

If **SnmpDecodeMsg** fails, the return value will be SNMPAPI_FAILURE and **SnmpGetLastError** will be set to report one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SESSION_INVALID	Indicates that a session parameter is invalid.
SNMPAPI_ENTITY_INVALID	Indicates that an entity parameter is invalid.
SNMPAPI_CONTEXT_INVALID	Indicates that the context parameter is invalid.
SNMPAPI_PDU_INVALID	Indicates that the PDU parameter is invalid.
SNMPAPI_OUTPUT_TRUNCATED	Indicates that the buffer was too small. That is, "len" bytes of "ptr" were consumed before reaching the end of the encoded message; no output parameters are created.
SNMPAPI_MESSAGE_INVALID	The SNMP message described by the 'msgBufDesc' parameter is invalid; no output resources are created.

Comments:

The **SmnpDecodeMsg** function is meant to be symmetrical with the **SnmpEncodeMsg** function. Refer to **SnmpEncodeMsg** for additional insight into the operation and possible failure modes of the **SnmpDecodeMsg** function.

3.6.8. SnmpFreeDescriptor()

The **SnmpFreeDescriptor** function is used by the application to inform the implementation that it no longer requires access to a WinSNMP "descriptor object" that had been "populated" earlier on its behalf by the implementation.

Syntax:

```
SNMPAPI_STATUS     SnmpFreeDescriptor (
      IN smiUINT32          syntax,
      IN smiLPOPAQUE        descriptor);
```

Parameter	Description
syntax	Identifies the "syntax" (data type) of the target descriptor.
descriptor	Identifies the target descriptor object..

Returns:

Upon successful completion, **SnmpFreeDescriptor** returns SNMPAPI_SUCCESS.

If **SnmpFreeDescriptor** fails, the return value will be SNMPAPI_FAILURE and **SnmpGetLastError** will be set to report one of the following:

SnmpGetLastError()	Description
"Common Error Codes"	See Section 2.10.1. Common Error Codes.
SNMPAPI_SYNTAX_INVALID	Indicates that the syntax parameter is invalid for this function.
SNMPAPI_OPERATION_INVALID	Indicates that the descriptor parameter is invalid for this function.

Comments:

See Section 2.11.4. Descriptors.

WinSNMP "descriptor objects" are either smiOID or smiOCTETS structures (or equivalents, such as smiIPADDR and smiOPAQUE) and consist of a "len" member and a "ptr" member.

These objects are "populated" by the implementation on behalf of the application in response to any "OUT" parameter of type smiOID, smiOCTETS, and smiVALUE. (Note that any smiVALUE structure may or may not contain an smiOID or smiOCTETS structure in its "value" member upon return from **SnmpGetVb**, as will be indicated by the associated "syntax" member of the smiVALUE structure.) In addition to **SnmpGetVb**, the following functions also result in the implementation populating a descriptor object for the application: **SnmpContextToStr**, **SnmpStrToOid**, **SnmpOidCopy**, **SnmpEncodeMsg**. Others may be added later.

Note that applications should not attempt to free memory returned in the "ptr" member of descriptor objects that have been populated by the implementation. The method of memory allocation and, consequently, deallocation, for these variables is private to the implementation, and hidden from the application except for the **SnmpFreeDescriptor** interface described above.

Note that the "syntax" parameter can be used by implementations to distinguish among different varieties of descriptor objects, if necessary. The SNMPAPI_OPERATION_INVALID error can be returned if, for example, the "descriptor" parameter does not satisfy implementation-specific requirements (e.g., the implementation can recognize that the "ptr" member does not identify an allocation that it has made on behalf of the calling application or if the indicated allocation had already been released by the application in a prior call to **SnmpFreeDescriptor**).

4. Declarations

This section--exclusive of this preamble text--constitutes the WinSNMP.h *include* file containing common declarations for SNMP datatypes, attributes, and values and for WinSNMP API datatypes, attributes, and values. This section (minus this preamble text) must be delivered as WinSNMP.h with every compliant implementation.

Additional declarations required or offered by an implementation must be delivered in a separate include file with an implementation-specific name.

An attempt has been made to balance brevity and clarity in these declarations. In general, however, there has been a slight bias toward brevity. Developers can easily include longer, more descriptive equivalents to the declarations through additional *#define* and *typedef* statements in a private *include* files loaded after WinSNMP.h.

```
/* v1.1 WinSNMP.h */
/* v1.0 - Sep 13, 1993 */
/* v1.1 - Jun 10, 1994 */

#ifndef _INC_WINSNMP        /* Include WinSNMP declarations */
#define  _INC_WINSNMP       /* Just once! */

#ifndef _INC_WINDOWS        /* Include Windows declarations, if not already done */
#include <windows.h>
#define _INC_WINDOWS        /* Just once! */
#endif                      /* _INC_WINDOWS */

#ifdef __cplusplus
extern "C" {
#endif

/* WinSNMP API Type Definitions  */
typedef  HANDLE             HSNMP_SESSION,    FAR *LPHSNMP_SESSION;
typedef  HANDLE             HSNMP_ENTITY,     FAR *LPHSNMP_ENTITY;
typedef  HANDLE             HSNMP_CONTEXT,    FAR *LPHSNMP_CONTEXT;
typedef  HANDLE             HSNMP_PDU,        FAR *LPHSNMP_PDU;
typedef  HANDLE             HSNMP_VBL,        FAR *LPHSNMP_VBL;
typedef unsigned char       smiBYTE,          FAR *smiLPBYTE;
/* SNMP-related types from RFC1442 (SMI) */
typedef signed long         smiINT,           FAR *smiLPINT;
typedef smiINT              smiINT32,         FAR *smiLPINT32;
typedef unsigned long       smiUINT32,        FAR *smiLPUINT32;
typedef struct {
    smiUINT32 len;
    smiLPBYTE ptr;}         smiOCTETS,        FAR *smiLPOCTETS;
typedef const smiOCTETS                       FAR *smiLPCOCTETS;
typedef smiOCTETS           smiBITS,          FAR *smiLPBITS;
typedef struct {
    smiUINT32   len;
    smiLPUINT32 ptr;}       smiOID,           FAR *smiLPOID;
typedef const smiOID                          FAR *smiLPCOID;
typedef smiOCTETS           smiIPADDR,        FAR *smiLPIPADDR;
typedef smiUINT32           smiCNTR32,        FAR *smiLPCNTR32;
typedef smiUINT32           smiGAUGE32,       FAR *smiLPGAUGE32;
typedef smiUINT32           smiTIMETICKS,     FAR *smiLPTIMETICKS;
typedef smiOCTETS           smiOPAQUE,        FAR *smiLPOPAQUE;
typedef smiOCTETS           smiNSAPADDR,      FAR *smiLPNSAPADDR;
typedef struct {
      smiUINT32 hipart;
      smiUINT32 lopart;}    smiCNTR64,        FAR *smiLPCNTR64;

/* ASN/BER Base Types */
/* (used in forming SYNTAXes and certain SNMP types/values) */
#define ASN_UNIVERSAL           (0x00)
#define ASN_APPLICATION         (0x40)
#define ASN_CONTEXT             (0x80)
#define ASN_PRIVATE             (0xC0)
#define ASN_PRIMITIVE           (0x00)
#define ASN_CONSTRUCTOR         (0x20)
```

```
/* SNMP ObjectSyntax Values */
#define SNMP_SYNTAX_SEQUENCE  (ASN_UNIVERSAL | ASN_CONSTRUCTOR | 0x10)
/* These values are used in the "syntax" member of the smiVALUE structure which follows */
#define SNMP_SYNTAX_INT           (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x02)
#define SNMP_SYNTAX_BITS          (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x03)
#define SNMP_SYNTAX_OCTETS        (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x04)
#define SNMP_SYNTAX_NULL          (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x05)
#define SNMP_SYNTAX_OID           (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x06)
#define SNMP_SYNTAX_INT32         SNMP_SYNTAX_INT
#define SNMP_SYNTAX_IPADDR        (ASN_APPLICATION | ASN_PRIMITIVE | 0x00)
#define SNMP_SYNTAX_CNTR32        (ASN_APPLICATION | ASN_PRIMITIVE | 0x01)
#define SNMP_SYNTAX_GAUGE32       (ASN_APPLICATION | ASN_PRIMITIVE | 0x02)
#define SNMP_SYNTAX_TIMETICKS     (ASN_APPLICATION | ASN_PRIMITIVE | 0x03)
#define SNMP_SYNTAX_OPAQUE        (ASN_APPLICATION | ASN_PRIMITIVE | 0x04)
#define SNMP_SYNTAX_NSAPADDR      (ASN_APPLICATION | ASN_PRIMITIVE | 0x05)
#define SNMP_SYNTAX_CNTR64        (ASN_APPLICATION | ASN_PRIMITIVE | 0x06)
#define SNMP_SYNTAX_UINT32        (ASN_APPLICATION | ASN_PRIMITIVE | 0x07)
/* Exception conditions in response PDUs for SNMPv2 */
#define SNMP_SYNTAX_NOSUCHOBJECT     (ASN_CONTEXT | ASN_PRIMITIVE | 0x00)
#define SNMP_SYNTAX_NOSUCHINSTANCE   (ASN_CONTEXT | ASN_PRIMITIVE | 0x01)
#define SNMP_SYNTAX_ENDOFMIBVIEW     (ASN_CONTEXT | ASN_PRIMITIVE | 0x02)

typedef struct {                        /* smiVALUE portion of VarBind */
        smiUINT32       syntax;         /* Insert SNMP_SYNTAX_<type> */
        union {
        smiINT          sNumber;        /* SNMP_SYNTAX_INT
                                           SNMP_SYNTAX_INT32 */
        smiUINT32       uNumber;        /* SNMP_SYNTAX_UINT32
                                           SNMP_SYNTAX_CNTR32
                                           SNMP_SYNTAX_GAUGE32
                                           SNMP_SYNTAX_TIMETICKS */
        smiCNTR64       hNumber;        /* SNMP_SYNTAX_CNTR64 */
        smiOCTETS       string;         /* SNMP_SYNTAX_OCTETS
                                           SNMP_SYNTAX_BITS
                                           SNMP_SYNTAX_OPAQUE
                                           SNMP_SYNTAX_IPADDR
                                           SNMP_SYNTAX_NSAPADDR */
        smiOID          oid;            /* SNMP_SYNTAX_OID */
        smiBYTE         empty;          /* SNMP_SYNTAX_NULL
                                           SNMP_SYNTAX_NOSUCHOBJECT
                                           SNMP_SYNTAX_NOSUCHINSTANCE
                                           SNMP_SYNTAX_ENDOFMIBVIEW */
             }          value;          /* union */
        }               smiVALUE,       FAR *smiLPVALUE;
typedef const smiVALUE                  FAR *smiLPCVALUE;

/* SNMP Limits  */
#define MAXOBJIDSIZE        128     /* Max number of components in an OID */
#define MAXOBJIDSTRSIZE     1408    /* Max len of decoded MAXOBJIDSIZE OID */

/* PDU Type Values */
#define SNMP_PDU_GET              (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x0)
#define SNMP_PDU_GETNEXT          (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x1)
#define SNMP_PDU_RESPONSE         (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x2)
#define SNMP_PDU_SET              (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x3)
/* SNMP_PDU_V1TRAP is obsolete in SNMPv2 */
#define SNMP_PDU_V1TRAP           (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x4)
#define SNMP_PDU_GETBULK          (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x5)
#define SNMP_PDU_INFORM           (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x6)
#define SNMP_PDU_TRAP             (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x7)
```

```
/* SNMPv1 Trap Values */
/* (These values might be superfluous wrt WinSNMP applications) */
#define SNMP_TRAP_COLDSTART                 0
#define SNMP_TRAP_WARMSTART                 1
#define SNMP_TRAP_LINKDOWN                  2
#define SNMP_TRAP_LINKUP                    3
#define SNMP_TRAP_AUTHFAIL                  4
#define SNMP_TRAP_EGPNEIGHBORLOSS           5
#define SNMP_TRAP_ENTERPRISESPECIFIC        6

/* SNMP Error Codes Returned in Error_status Field of PDU */
/* (these are NOT WinSNMP API Error Codes */
/* Error Codes Common to SNMPv1 and SNMPv2 */
#define SNMP_ERROR_NOERROR                  0
#define SNMP_ERROR_TOOBIG                   1
#define SNMP_ERROR_NOSUCHNAME               2
#define SNMP_ERROR_BADVALUE                 3
#define SNMP_ERROR_READONLY                 4
#define SNMP_ERROR_GENERR                   5
/* Error Codes Added for SNMPv2 */
#define SNMP_ERROR_NOACCESS                 6
#define SNMP_ERROR_WRONGTYPE                7
#define SNMP_ERROR_WRONGLENGTH              8
#define SNMP_ERROR_WRONGENCODING            9
#define SNMP_ERROR_WRONGVALUE               10
#define SNMP_ERROR_NOCREATION               11
#define SNMP_ERROR_INCONSISTENTVALUE        12
#define SNMP_ERROR_RESOURCEUNAVAILABLE      13
#define SNMP_ERROR_COMMITFAILED             14
#define SNMP_ERROR_UNDOFAILED               15
#define SNMP_ERROR_AUTHORIZATIONERROR       16
#define SNMP_ERROR_NOTWRITABLE              17
#define SNMP_ERROR_INCONSISTENTNAME         18

/* WinSNMP API Values */
/* Values used to indicate entity/context translation modes */
#define SNMPAPI_TRANSLATED              0
#define SNMPAPI_UNTRANSLATED_V1         1
#define SNMPAPI_UNTRANSLATED_V2         2

/* Values used to indicate SNMP "communications  level" supported by the implementation */
#define SNMPAPI_NO_SUPPORT              0
#define SNMPAPI_V1_SUPPORT              1
#define SNMPAPI_V2_SUPPORT              2
#define SNMPAPI_M2M_SUPPORT             3

/* Values used to indicate retransmit mode in the implementation */
#define SNMPAPI_OFF                     0       /* Refuse support */
#define SNMPAPI_ON                      1       /* Request support */

/* WinSNMP API Function Return Codes */
typedef smiUINT32           SNMPAPI_STATUS;     /* Used for function ret values */
#define SNMPAPI_FAILURE                 0       /* Generic error code */
#define SNMPAPI_SUCCESS                 1       /* Generic success code */
```

```
/* WinSNMP API Error Codes (for SnmpGetLastError) */
/* (NOT SNMP Response-PDU error_status codes) */
#define SNMPAPI_ALLOC_ERROR             2       /* Error allocating memory */
#define SNMPAPI_CONTEXT_INVALID         3       /* Invalid context parameter */
#define SNMPAPI_CONTEXT_UNKNOWN         4       /* Unknown context parameter */
#define SNMPAPI_ENTITY_INVALID          5       /* Invalid entity parameter */
#define SNMPAPI_ENTITY_UNKNOWN          6       /* Unknown entity parameter */
#define SNMPAPI_INDEX_INVALID           7       /* Invalid VBL index parameter */
#define SNMPAPI_NOOP                    8       /* No operation performed */
#define SNMPAPI_OID_INVALID             9       /* Invalid OID parameter */
#define SNMPAPI_OPERATION_INVALID       10      /* Invalid/unsupported operation */
#define SNMPAPI_OUTPUT_TRUNCATED        11      /* Insufficient output buf len */
#define SNMPAPI_PDU_INVALID             12      /* Invalid PDU parameter */
#define SNMPAPI_SESSION_INVALID         13      /* Invalid session parameter */
#define SNMPAPI_SYNTAX_INVALID          14      /* Invalid syntax in smiVALUE */
#define SNMPAPI_VBL_INVALID             15      /* Invalid VBL parameter */
#define SNMPAPI_MODE_INVALID            16      /* Invalid mode parameter */
#define SNMPAPI_SIZE_INVALID            17      /* Invalid size/length parameter */
#define SNMPAPI_NOT_INITIALIZED         18      /* SnmpStartup failed/not called */
#define SNMPAPI_MESSAGE_INVALID         19      /* Invalid SNMP message format */
#define SNMPAPI_HWND_INVALID            20      /* Invalid Window handle */
#define SNMPAPI_OTHER_ERROR             99      /* For internal/undefined errors */
/* Generic Transport Layer (TL) Errors */
#define SNMPAPI_TL_NOT_INITIALIZED      100     /* TL not initialized */
#define SNMPAPI_TL_NOT_SUPPORTED        101     /* TL does not support protocol */
#define SNMPAPI_TL_NOT_AVAILABLE        102     /* Network subsystem has failed */
#define SNMPAPI_TL_RESOURCE_ERROR       103     /* TL resource error */
#define SNMPAPI_TL_UNDELIVERABLE        104     /* Destination unreachable */
#define SNMPAPI_TL_SRC_INVALID          105     /* Source endpoint invalid */
#define SNMPAPI_TL_INVALID_PARAM        106     /* Input parameter invalid */
#define SNMPAPI_TL_IN_USE               107     /* Source endpoint in use */
#define SNMPAPI_TL_TIMEOUT              108     /* No response before timeout */
#define SNMPAPI_TL_PDU_TOO_BIG          109     /* PDU too big for send/receive */
#define SNMPAPI_TL_OTHER                199     /* Undefined TL error */

/* WinSNMP API Function Prototypes */
#define IN                                      /* Documentation only */
#define OUT                                     /* Documentation only */
#define SNMPAPI_CALL        WINAPI              /* FAR PASCAL calling conventions */
```

```
/* Local Database Functions */

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetTranslateMode
                                  (OUT smiLPUINT32 nTranslateMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetTranslateMode
                                  (IN smiUINT32 nTranslateMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetRetransmitMode
                                  (OUT smiLPUINT32 nRetransmitMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetRetransmitMode
                                  (IN smiUINT32 nRetransmitMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetTimeout
                                  (IN HSNMP_ENTITY hEntity,
                                  OUT smiLPTIMETICKS nPolicyTimeout,
                                  OUT smiLPTIMETICKS nActualTimeout);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetTimeout
                                  (IN HSNMP_ENTITY hEntity,
                                  IN smiTIMETICKS nPolicyTimeout);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetRetry
                                  (IN HSNMP_ENTITY hEntity,
                                  OUT smiLPUINT32 nPolicyRetry,
                                  OUT smiLPUINT32 nActualRetry);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetRetry
                                  (IN HSNMP_ENTITY hEntity,
                                  IN smiUINT32 nPolicyRetry);
```

```
/* Communications Functions */

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpStartup
                                        (OUT smiLPUINT32 nMajorVersion,
                                        OUT smiLPUINT32 nMinorVersion,
                                        OUT smiLPUINT32 nLevel,
                                        OUT smiLPUINT32 nTranslateMode,
                                        OUT smiLPUINT32 nRetransmitMode);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpCleanup
                                        (void);

HSNMP_SESSION  SNMPAPI_CALL    SnmpOpen
                                        (IN HWND hWnd,
                                        IN UINT wMsg);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpClose
                                        (IN HSNMP_SESSION session);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpSendMsg
                                        (IN HSNMP_SESSION session,
                                        IN HSNMP_ENTITY srcEntity,
                                        IN HSNMP_ENTITY dstEntity,
                                        IN HSNMP_CONTEXT context,
                                        IN HSNMP_PDU PDU);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpRecvMsg
                                        (IN HSNMP_SESSION session,
                                        OUT LPHSNMP_ENTITY srcEntity,
                                        OUT LPHSNMP_ENTITY dstEntity,
                                        OUT LPHSNMP_CONTEXT  context,
                                        OUT LPHSNMP_PDU PDU);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpRegister
                                        (IN HSNMP_SESSION session,
                                        IN HSNMP_ENTITY srcEntity,
                                        IN HSNMP_ENTITY dstEntity,
                                        IN HSNMP_CONTEXT context,
                                        IN smiLPCOID notification,
                                        IN smiUINT32 state);
```

```
/* Entity/Context Functions */

HSNMP_ENTITY        SNMPAPI_CALL        SnmpStrToEntity
                                        (IN HSNMP_SESSION session,
                                        IN LPCSTR string);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpEntityToStr
                                        (IN HSNMP_ENTITY entity,
                                        IN smiUINT32 size,
                                        OUT LPSTR string);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpFreeEntity
                                        (IN HSNMP_ENTITY entity);

HSNMP_CONTEXT       SNMPAPI_CALL        SnmpStrToContext
                                        (IN HSNMP_SESSION session,
                                        IN smiLPCOCTETS string);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpContextToStr
                                        (IN HSNMP_CONTEXT context,
                                        OUT smiLPOCTETS string);

SNMPAPI_STATUS      SNMPAPI_CALL        SnmpFreeContext
                                        (IN HSNMP_CONTEXT context);
```

```
/* PDU Functions */

HSNMP_PDU           SNMPAPI_CALL    SnmpCreatePdu
                                    (IN HSNMP_SESSION session,
                                    IN smiINT PDU_type,
                                    IN smiINT32 request_id,
                                    IN smiINT error_status,
                                    IN smiINT error_index,
                                    IN HSNMP_VBL varbindlist);

SNMPAPI_STATUS      SNMPAPI_CALL    SnmpGetPduData
                                    (IN HSNMP_PDU PDU,
                                    OUT smiLPINT PDU_type,
                                    OUT smiLPINT32 request_id,
                                    OUT smiLPINT error_status,
                                    OUT smiLPINT error_index,
                                    OUT LPHSNMP_VBL varbindlist);

SNMPAPI_STATUS      SNMPAPI_CALL    SnmpSetPduData
                                    (IN HSNMP_PDU PDU,
                                    IN const smiINT FAR *PDU_type,
                                    IN const smiINT32 FAR *request_id,
                                    IN const smiINT FAR *non_repeaters,
                                    IN const smiINT FAR *max_repetitions,
                                    IN const HSNMP_VBL FAR *varbindlist);

HSNMP_PDU           SNMPAPI_CALL    SnmpDuplicatePdu
                                    (IN HSNMP_SESSION session,
                                    IN HSNMP_PDU PDU);

SNMPAPI_STATUS      SNMPAPI_CALL    SnmpFreePdu
                                    (IN HSNMP_PDU PDU);
```

```
/* Variable-Binding Functions */

HSNMP_VBL           SNMPAPI_CALL    SnmpCreateVbl
                                    (IN HSNMP_SESSION session,
                                    IN smiLPCOID name,
                                    IN smiLPCVALUE value);

HSNMP_VBL           SNMPAPI_CALL    SnmpDuplicateVbl
                                    (IN HSNMP_SESSION session,
                                    IN HSNMP_VBL vbl);

SNMPAPI_STATUS      SNMPAPI_CALL    SnmpFreeVbl
                                    (IN HSNMP_VBL vbl);

SNMPAPI_STATUS      SNMPAPI_CALL    SnmpCountVbl
                                    (IN HSNMP_VBL vbl);

SNMPAPI_STATUS      SNMPAPI_CALL    SnmpGetVb
                                    (IN HSNMP_VBL vbl,
                                    IN smiUINT32 index,
                                    OUT smiLPOID name,
                                    OUT smiLPVALUE value);

SNMPAPI_STATUS      SNMPAPI_CALL    SnmpSetVb
                                    (IN HSNMP_VBL vbl,
                                    IN smiUINT32 index,
                                    IN smiLPCOID name,
                                    IN smiLPCVALUE value);

SNMPAPI_STATUS      SNMPAPI_CALL    SnmpDeleteVb
                                    (IN HSNMP_VBL vbl,
                                    IN smiUINT32 index);
```

```
/* Utility Functions */

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetLastError
                                  (IN HSNMP_SESSION session);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpStrToOid
                                  (IN LPCSTR string,
                                  OUT smiLPOID dstOID);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpOidToStr
                                  (IN smiLPCOID srcOID,
                                  IN smiUINT32 size,
                                  OUT LPSTR string);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpOidCopy
                                  (IN smiLPCOID srcOID,
                                  OUT smiLPOID dstOID);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpOidCompare
                                  (IN smiLPCOID xOID,
                                  IN smiLPCOID yOID,
                                  IN smiUINT32 maxlen,
                                  OUT smiLPINT result);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpEncodeMsg
                                  (IN HSNMP_SESSION session,
                                  IN HSNMP_ENTITY srcEntity,
                                  IN HSNMP_ENTITY dstEntity,
                                  IN HSNMP_CONTEXT context,
                                  IN HSNMP_PDU pdu,
                                  OUT smiLPOCTETS msgBufDesc);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpDecodeMsg
                                  (IN HSNMP_SESSION session,
                                  OUT LPHSNMP_ENTITY srcEntity,
                                  OUT LPHSNMP_ENTITY dstEntity,
                                  OUT LPHSNMP_CONTEXT context,
                                  OUT LPHSNMP_PDU pdu,
                                  IN smiLPCOCTETS msgBufDesc);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpFreeDescriptor
                                  (IN smiUINT32 syntax,
                                  IN smiLPOPAQUE descriptor);

#ifdef __cplusplus
}
#endif
#endif            /* _INC_WINSNMP */
```

Appendix A. Mapping Traps Between SNMPv1 and SNMPv2

One of the differences between SNMPv1 and SNMPv2 is a change to the trap format: In SNMPv1, the trap format was unlike the format of the other protocol data units; in SNMPv2 the trap format is identical to the format of the other PDUs.

When the WinSNMP API delivers a trap to a management application, it always uses the SNMPv2 trap format, even if an SNMPv1 agent generated the trap. The SNMPv2 coexistence document, RFC 1452, specifies how an SNMPv1 trap is translated into the SNMPv2 trap format, and this algorithm is used in all implementations of the WinSNMP API.

In SNMPv1, the trap format has five fields:

- enterprise, which identified the type of device which generated the trap;
- agent-addr, which identified the network address of the device;
- generic-trap/specific-trap, which identified the trap which was generated;
- time-stamp, which identified when the trap was generated; and,
- variable-bindings, which contained the "payload", if any, associated with the trap.

In SNMPv2, the trap format consists simply of a list of "N" variable bindings, in which:

- the first variable binding contains the time-stamp;
- the second variable binding identifies the trap, using an OBJECT IDENTIFIER;
- the third through "N" variable bindings, if any, contain the payload.

Hence, when **SnmpRecvMsg** returns a message whose operation type is SNMP_PDU_TRAP, the application need only examine the variable-bindings of that message in order to ascertain the information associated with the trap.

When translating an SNMPv1 trap to the SNMPv2 format, one additional variable binding may be present, at the end of the list, which corresponds to the enterprise field. According to the SNMPv2 coexistence document, this variable binding need only be present if the trap was enterprise-specific. However, in order to simplify the programming of management applications, this variable binding is always added by the WinSNMP API when it translates an SNMPv1 trap to the SNMPv2 format.

The following code fragment shows how an application can examine the variable-bindings in order to retrieve all of the information associated with a trap.

```
SNMPAPI_STATUS TrapProcess (HSNMP_SESSION hSession)
{
      HSNMP_ENTITY hSrc, hDest;
      HSNMP_CONTEXT hContext;
      HSNMP_PDU hPDU;
      HSNMP_VBL hVBL;
      smiINT32 Request_id;
      smiINT PduType, Err_stat, Err_index;
      SNMPAPI_STATUS RetStatus, Index, VBCount;
      smiOID   Name;
      smiVALUE Value;
      smiBYTE  NameBuffer[100], ValueBuffer[256];
      ...
      Request_id = SnmpRecvMsg (
            hSession,            // Trap Session Handle
            &hSrc,               // Source Entity Handle
            &hDest,              // Destination Entity Handle
            &hContext,           // Context Handle
            &hPDU);              // PDU Handle
      // Error condition checking for SnmpRecvMsg() performs here.

      RetStatus = SnmpGetPduData (
            hPDU,                // PDU Handle
            &PduType,            // PDU return type
            &Request_id,         // Request ID of the Trap
            &Err_stat,           // Error status for a variable
            &Err_index,          // Index to the variable with error
            &hVBL);              // Handle to the Varbindlist

      // Sample error checking for SnmpGetPduData():
      if ((RetStatus == SNMPAPI_FAILURE) ||
         (PduType != SNMP_PDU_TRAP) ||
         (Err_stat != SNMP_ERROR_NOERROR)) {
            SnmpFreePdu(hPDU);
            SnmpFreeEntity(hSrc);
            SnmpFreeEntity(hDest);
            SnmpFreeContext(hContext);
            return (SnmpGetLastError(hSession));
      }

      VBCount = SnmpCountVbl(hVBL);
      for (Index = 1; i <= VBCount; Index ++) {

            // When Index = 1,
            //  Oid   = sysUpTimeOid
            //  Value = uptime value for the V1 time-stamp trap field
            // When Index = 2,
            //  Oid   = v2snmpTrapOid
            //  value = can be one of the following Oids:
            //            v2coldStartOid
            //            v2warmStartOid
            //            v2linkDownOid
            //            v2linkUpOid
            //            v2authenFailureOid
            //            v2egpNeighborLossOid
            //            v2snmpTrapEnterpriseOid+specific_trap
            // When Index = VBCount, (the last Oid in the v2 trap)
            //  Oid   = v2snmpTrapEnterpriseOid
            //  Value = enterprise specific Oid from V1 trap
```

```
            // Get a particular variable from the Varbindlist
            // using the given Index.

              RetStatus = SnmpGetVb (
                  hVBL,              // Input Varbindlist Handle
                  Index,             // Index to a variable
                  &Name,             // Output name of the variable
                  &Value);           // Output value of the variable
              // Error condition checking for RetStatus performs here

              SnmpOidToStr (&Name, 100, (LPSTR)NameBuffer);
              SnmpFreeDescriptor (SNMP_SYNTAX_OID, &Name);

              switch (Value.syntax) {

                    case SNMP_SYNTAX_INT :
                    _ltoa ((long)Value.value.sNumber, ValueBuffer, 10);
                    break;

                    case SNMP_SYNTAX_UINT32 :
                    case SNMP_SYNTAX_CNTR32 :
                    case SNMP_SYNTAX_GAUGE32 :
                    case SNMP_SYNTAX_TIMETICKS :
                    _ltoa ((long)Value.value.uNumber, ValueBuffer, 10);
                    break;

                    case SNMP_SYNTAX_CNTR64 :
                    break;    // Need routine to convert 64-bit number to string here!

                    case SNMP_SYNTAX_OCTETS :
                    case SNMP_SYNTAX_BITS :
                    case SNMP_SYNTAX_OPAQUE :
                    case SNMP_SYNTAX_IPADDR :
                    case SNMP_SYNTAX_NSAPADDR :
                    _fmemcpy (ValueBuffer, Value.value.string.ptr, (size_t)Value.value.string.len);
                    SnmpFreeDescriptor (SNMP_SYNTAX_OCTETS, &Value.value.string);
                    break;

                    case SNMP_SYNTAX_OID :
                    SnmpOidToStr (&Value.value.oid, 256, ValueBuffer);
                    SnmpFreeDescriptor (SNMP_SYNTAX_OID, &Value.value.oid);
                    break;

              } // switch

              OutputVariable (
                    Index,             // Index to a given variable for output
                    NameBuffer,        // Trap variable Oid Name Output Buffer
                    ValueBuffer);      // Trap variable Value Output Buffer
        } //for loop

        SnmpFreeEntity (hSrc);
        SnmpFreeEntity (hDest);
        SnmpFreeContext (hContext);
        SnmpFreeVbl (hVBL);
        SnmpFreePdu (hPDU);

        return SNMPAPI_SUCCESS;
} // TrapProcess
```

Appendix B. Usage Example

Sample applications will be published separately due to size and packaging considerations.

Appendix C. WinSNMP++ Prototype

The following table shows a possible mapping between the C interface of the WinSNMP API specification and a set of C++ wrappers. This is strictly an informational Appendix and does not constitute a part of the official Windows SNMP API specification at this time.

The content was contributed by Maria Greene, and was well-discussed on the list. Unless otherwise noted, obvious differences between the WinSNMP C interface specifications and the WinSNMP++ interface prototypes shown below are simply due to changes being made to the C interface after this Appendix was inserted. Therefore, future edits for consistency--according to the pattern demonstrated herein--can be assumed.

Please note the re-positioning of the varbindlist parameter in the SnmpCreatePdu and SnmpSetPduData C++ equivalents is intentional and permits applications to take advantage of certain C++ language-specific features.

Class/Returns	C++ Interface	C Interface
Session::	Session (Entity &srcEntity, HWND hWnd, UINT wMsg)	SnmpOpen
	~Session (void)	SnmpClose
BOOL	SendMsg (const Entity &dstEntity, const Context &context, const Pdu &PDU)	SnmpSendMsg
Pdu *	RecvMsg (Entity &srcEntity, Entity &dstEntity, Context &context)	SnmpRecvMsg
BOOL	Register (const Entity &entity, const Context &context, const Oid &notification = NULL, BOOL enabled = TRUE)	SnmpRegister
Entity::	Entity (const LPSTR entity) Entity (const Entity &entity)	SnmpStrToEntity
	~Entity (void)	SnmpFreeEntity
Entity	operator const char*() const &operator = (const Entity &entity)	SnmpEntityToStr
Context::	Context (const LPSTR context) Context (const Context &context)	SnmpStrToContext
	~Context (void)	SnmpFreeContext
Context	operator const char*() const &operator = (const Context &context)	SnmpContextToStr
SnmpUtil::	SnmpUtil (???)	SnmpStartup
	~SnmpUtil (void)	SnmpCleanup
????	EnumEntities (???)	SnmpEnumEntities
????	EnumContexts (???)	SnmpEnumContexts
int	GetLastError (void)	SnmpGetLastError
DWORD	Version (void)	SnmpUtilVersion

Class/Returns	C++ Interface	C Interface
Pdu::	Pdu (smiINT PDU_type= SNMP_PDU_GETNEXT, VarBindList *varbindlist = NULL, smiINT32 request_id = 0, smiINT non_repeaters = 0, smiINT max_repetitions = 0)	SnmpCreatePdu
	Pdu (const Pdu &PDU)	SnmpDuplicatePdu
	~Pdu (void)	SnmpFreePdu
BOOL	GetData (smiLPINT PDU_type = NULL, smiLPINT32 request_id = NULL, smiLPINT error_status = NULL, smiLPINT error_index = NULL, VarBindList **varbindlist = NULL)	SnmpGetPduData
BOOL	SetData (smiINT PDU_type = SNMP_PDU_GETNEXT, VarBindList *pvarbindlist = NULL, smiINT32 request_id = 0, smiINT non_repeaters = 0, smiINT max_repetitions = 0)	SnmpSetPduData
Pdu	&operator = (const Pdu &PDU)	
VarBindList::	VarBindList (Oid *poid = NULL, smiLPVALUE value = NULL)	SnmpCreateVbl
	VarBindList (&VarBindList)	SnmpDuplicateVbl
	~VarBindList (void)	SnmpFreeVbl
smiINT	Count (void)	SnmpCountVbl
BOOL	GetVb (smiINT index, Oid &oid, smiLPVALUE value)	SnmpGetVb
BOOL	SetVb (Oid &oid, smiLPVALUE value)	SnmpSetVb
BOOL	DeleteVb (Oid &oid)	SnmpDeleteVb
VarBindList	&operator = (const VarBindList &varbindlist)	
Oid::	Oid (LPSTR str = NULL)	SnmpUtilStrToOid
	Oid (const Oid &oid)	SnmpUtilOidCpy
	~Oid (void)	
	operator const char*() const	SnmpUtilOidToStr
Oid	&operator += (const Oid &oid)	SnmpUtilOidAppend
BOOL	operator == (const Oid &oid) const	SnmpUtilOidCmp
Oid	&operator = (const LPSTR str)	SnmpUtilStrToOid
Oid	&operator = (const Oid &oid)	SnmpUtilOidCpy
smiINT	NCmp (const Oid &oid, smiUINT32 len = 0) const	SnmpUtilOidNCmp