Windows SNMP Agent

The Open Interface for Programming the Extensible SNMP V1 and V2 Agent Under Microsoft® Windows™

Version 0.3

1 December 1994

Aleksey Romanov	Paul Freeman Associates, Inc
William H. White	Digital Equipment Corporation
Pete Wilson	Paul Freeman Associates, Inc

14 Pleasant Street P. O. Box 2067
Westford, Massachusetts 01886-5067

Comments and questions may be submitted by electronic mail to winsnmp@mailbag.intel.com. Requests to be added to the WinSNMP mailing list should be addressed as:

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

The binary of the WinSNMP Agent demonstrates the feasilbility of this specification. You can obtain this form of the Agent via anonymous FTP. The demonstration package includes an implementation of the core Agent for the SNMP v1 and v2; the SNMP and System groups of MIB-II; and the required and hrSWRun groups from the Host Resources MIB (RFC 1514). To FTP this binary:

ftp ftp.std.com // 192.74.137.7
user ftp *or* anonymous
pass your e-mail address
cd vendors/snmp/windows-agent
binary
Then mget all of the files in that directory. The README file gives further instructions.

Authors' Contact Information

Aleksey Romanov	ralex@world.std.com
William H. White	b_white@ranger.enet.dec.com
Pete Wilson	pwilson@world.std.com

Version history and status of this version

0.1	1 Jan 1994	Original offering.
0.2	1 Sep 1994	1. Version 0.2 reflects changes made during the implementation of the WinSNMP Agent and its accompanying MIBs (pw). 2. Added feature allowing overlapping MIBs/namespaces (pw).
0.3	1 Dec 1994	Corrected winsnmp mail-list subscription info; add support for hrSWRun Group in HR MIB accompanying WESA, the Windows Extensible SNMP Agent.

1 INTRODUCTION

1.1 Identification

This paper is the WinSNMP/Agent Specification, Version 0.2. It suggests the rules for the implementation of extensible, interoperable, vendor-neutral SNMP agent software that observes the rules of the SNMP Version 1 and Version 2 and that operates under the Microsoft® Windows™ family of operating systems.

1.2 Trademarks

Microsoft, MS, and MS-DOS are registered trademarks; and Windows is a trademark of Microsoft Corporation.

The Universal SNMP v1+v2 Agent and Open Agent Architecture are trademarks of Paul Freeman Associates, Inc.

1.3 Background

The architecture described in this paper is the Open Agent Architecture developed as part of the Universal SNMP v1+v2 Agent by Paul Freeman Associates, Inc., which makes the technology available to the WinSNMP/Agent effort.

Paul Freeman Associates, Inc., is the developer of the Universal SNMP v1+v2 Agent ™, a commercial implementation of the agent part of the SNMP. In order to advance the state of the network-management art, PFA is making available to the WinSNMP/Agent effort the part of its Agent which realizes the interface between the core part of its Agent and MIB implementation. It is this interface which enables independent third-party MIBs to operate correctly with the PFA core Agent. The PFA interface is referred to below as the "Interface."

The impetus for the revelation of this trade-secret material is the recent effort to standardize SNMP components for operation under the Microsoft Windows operating system: because the Interface has proven useful and complete, PFA offers it as the interface for the emergent Win/SNMP Agent standard.

1.4 Goals of This Paper

1.4.1 The paper describes a robust, powerful, and understandable framework for the implementation of extensible, interoperable, vendor-neutral Windows-based SNMP agents.

1.4.2 The paper defines a standard interface between the WinSNMP's core agent and its MIB-server modules and thereby supports the independent development of core agents and MIB servers.

1.4.3 The paper encourages the development of interoperable Windows-based agent components by different vendors so as to allow the dynamic selection of core-agent and MIB-server configurations by the end user from a set of fully interoperable core-agent and MIB-server components.

1.4.4 The paper describes standardized elements which are focused enough to permit the creation of a broad range of product-specific, value-added agents by conforming yet competing vendors, but proposes no further policy.

1.4.5 The paper hopes to offer APIs that are as compatible and consistent as possible with the WinSNMP/Manager and WinSNMP/MIB APIs.

1.4.6 The paper proposes a DMTF interface.

1.5 Non-Goals of This Paper

1.5.1 The paper sets no implementation policy, but only presents an implementation framework.

1.5.2 The paper is silent on issues of correct and compliant SNMP-agent development.

1.5.3 The paper proposes no communications scheme or transport layer of any kind, except to say by implication that some (conceptual) transport layer is present. The choice of transport layer and its realization is entirely the province of each agent implementor.

1.6 What is the WinSNMP/Agent?

The WinSNMP/Agent is an extensible SNMPv1- and SNMPv2-compliant agent that runs under Microsoft Windows. This specification describes a model for implementing the extensible agent. This model strictly partitions the agent into three parts:

- a protocol-specific part, called the "core agent";
- one or more MIB-specific parts, called "MIB servers"; and
- the interface between the two, called the "core-agent-to-MIB server interface" or, more often and more simply, the "interface" or the "API."

1.6.1 The WinSNMP/Agent Core Agent

A core agent is a Windows application that binds to a transport library for receiving or sending SNMP packets and for processing the SNMP v1/v2 PDU header. The core agent never accesses any MIB variables directly, but relies completely upon one or more MIB servers to perform MIB-variable accesses. The core agent completely manages SNMP-packet processing:

- it receives the SNMP request packet from the communications stack.
- it verifies the packet's authenticity, privacy, and context.
- it locates the appropriate MIB server for each variable in the packet.
- it passes each variable binding and protocol operation, in turn, to the appropriate MIB server.
- it manages the synchronization of multiple-phase (SET) and multiple-MIB-server (GETNEXT, GETBULK, and SET) processing.
- it receives the result of each protocol operation for each variable binding from the MIB server and, when appropriate, encodes the result in an SNMP response packet.
- it returns the complete response packet to the communications stack for delivery to the NMS.

The core agent also provides a number of service primitives for general use including, among possibly many others:

- trap sending.
- BER parsing.
- oid comparison.
- common RowStatus service.
- view evaluation.
- error logging.

1.6.2 The WinSNMP/Agent MIB Server

A MIB server is a DLL module or MS-Windows process that provides access to MIB variables at the request of and on behalf of the core agent. A MIB server implements all of the functions necessary to carry out all of the variable-access operations requested in any SNMP packet for one or more variables. A MIB server may operate on part of a MIB; on a complete MIB; or on multiple MIBs.

The MIB variables under the purview of a MIB server are said to comprise that MIB server's "namespace". Namespaces are ordered and contiguous sets of variable instances. Namespaces may overlap.

A MIB server is bound to a core agent in either of two ways:

- The MIB server can be a dynamically-linkable (loadable) DLL which can be loaded and then unloaded at run time. A MIB server bound in this way is said to be "loadable."
- The MIB server can exist as a process separate from the core agent. Such a MIB server communicates with the core agent using shared memory and messages and is said to be "external."

1.6.3 The WinSNMP/Agent Core-Agent-to-MIB-Server Interface

The core-agent-to-MIB-server interface consists of four primitive types:

- an initialization primitive that lets each MIB server initialize its namespaces;
- variable-access primitives, including primitives that support GET and SET operations;
- a synchronization primitive that tells a MIB server when packet processing is starting or is complete; and
- general housekeeping primitives for timekeeping and views maintenance.

This specification defines two general primitive implementations:

- Primitives which the core agent accesses through direct function calls. This is the scheme used for loadable MIB servers.
- Primitives which the core agent accesses via the combined use of shared memory and the messaging system native to the Windows environment. This is the scheme used for external MIB servers.

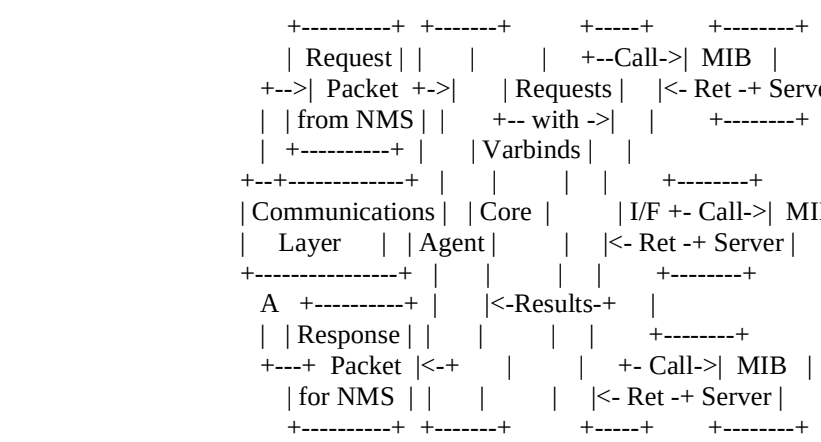

Figure 1.1 Control and data flow through some communications layer, the Core Agent, the Interface, and the MIB Server(s).

1.7 Naming Conventions

This section aims precisely to define terms which are local to this paper.

1.7.1 Mib

Each namespace (see just below) is represented to the core agent by one structure, called "mib" in this paper. There is a linked list of such mib structures, and each structure holds the complete definitions of:

- the single namespace under the control of the corresponding MIB server; and
- all MIB-server-resident primitives which operate on that namespace.

The core agent views all MIB servers mounted at any moment as a set of mib entries. The core agent has no interest whatever in the ways in which namespace access might be implemented within any MIB server.

1.7.2 Namespace

A namespace, equivalent to a range of MIB variables, is an ordered and contiguous set of variable instances accessible to one and only one MIB server. The set is characterized, defined, and bounded by a low boundary, a high boundary, and an entity name. The low boundary is that ASN.1 name which is less than or equal to the ASN.1 name of the first variable instance in the set. The high boundary is the least ASN.1 name which lies beyond the set ("beyond" in a lexicographic way). There is no variable instance whose ASN.1 name is greater than or equal to the low bound and less than the high bound which is accessed by any other MIB server. The entity name is the value of the appropriate contextLocalEntity.

1.7.3 Primitives

Each MIB server implements six callable functions, termed "primitives," which operate on its namespaces.

The MIB-server namespace primitives are:

- init() Initialize (one time) namespace
- indicate() Let the MIB server know packet processing is starting or is finished.
- look() Process GETs and phase one of SETs in the namespace.
- set() Process phase two of SETs in the namespace
- tick() Keep track of elapsed time.

There is no connection between the name of the primitive and the name of the function implementing the primitive operation. We use primitive names as function names in order to avoid another level of indirection.

For a loadable MIB server, the core agent calls the primitives using the function pointers associated with the namespace. These function pointers are stored in the namespace's mib structure. For an external MIB server, the core agent sets up shared memory and then passes messages to the server which cause the primitives to be invoked.

1.8 Glossary

This section glosses terms in general use in the SNMP and Windows communities.

<To be added>

2 OVERVIEW OF THE CORE-AGENT-TO-MIB-SERVER INTERFACE

This section is a brief, high-level overview of the "what" and "why" of the interface. A formal, detailed description of each interface component (the "how") appears later. This section concentrates on the interface as it exists for loadable MIB servers. The external MIB server interface, which depends upon messaging, is treated in detail below, but is functionally equivalent.

2.1 Major components

The main part of the interface is a set of six primitives: init(), indicate(), look(), set(), view(), and tick(). Each MIB server must offer this set of primitive functions.

2.1.1 Init Function

The init function for each namespace is called only once, and before any other MIB-server functions are called, to allow the MIB server to initialize itself and the namespaces it supports.

2.1.2 Indicate Function

The indicate function is called twice during the processing of each received SNMP packet. It is called, first, before calling any other mib primitives involved in processing the current received packet; and it's next called, second, after all varbinds in the request packet have been dealt with and the core agent is ready to return a response packet to the communications layer. The indicate function is called with an argument START or END, so will designate the first call as indicate(START) and the second and indicate(END).

The purpose of the function is to allow the MIB server to allocate and free resources synchronously with packet processing. For example, the MIB server might allocate temporary resources or locks at indicate(START) time and release them at indicate(END) time. The call also conveys information which is not going to change during the current packet processing: the version of this current request (SNMPv1 or SNMPv2), the type of operation performed, the reference (pointer or handle) of the view list, the index of the current view, and the temporal domain.

2.1.3 Look Function

The MIB server's look function must perform the search and access validation for the requested name. The details of the operations performed depend upon the pdu type being processed, pdu type established by indicate(START).

For the get (GetRequest pdu), the core agent supplies the name of the variable instance to find. The MIB server needs to find this variable instance, check access rights (the tools to do so having been delivered by indicate(START)), possibly copy this variable into some temporary static area, pass back its type and length (both unencoded) and a pointer to the variable, and return success or the appropriate error code.

The next (Get-Next or Get-Bulk pdu) transaction performs similarly, except that the name of the found variable is also returned.

The write (Set pdu) transaction differs from the two get transactions. In a write transaction, the core agent delivers the name of the variable to be set, its unencoded value, and the type and the length of the value. The MIB server then must find this variable instance, check access rights, check the value, the type, and the length provided; and then must store this value into shadow memory, along with all the information needed for a successful later low-level set operation. If any error is found, the MIB server returns the appropriate error code.

If the server finds that it can successfully process the request, the MIB server must return a non-zero 32-bit handle at least once for each packet for use by the core agent in subsequent set primitives.

2.1.4 Set Function

The core agent saves all (mib,handle,index) triplets for all non-zero handles passed back by the look primitive. Once the look primitive has been successfully performed for all varbinds in the current packet, the core agent core calls set(DO_PHASE1) for each stored (mib, handle,index) triplet. The MIB server now has the chance to verify the consistency of all variable instances associated with the handle value and/or all other variables in the server's namespaces (new values for all variables being known at this point), having made all arrangements necessary to insure a successful later commit. The MIB server returns an indication of success or the appropriate error code.

In case of error, the core agent, remembering the error code and the index, calls set(UNDO_PHASE1, handle, index) in reverse order for each stored and already processed triplet. The MIB server must undo all reservations made during the previous set(DO_PHASE1,...) calls.

If all calls to set(DO_PHASE1,...) were successfully performed, the core agent calls set(DO_COMMIT,handle,index) for all triplets. If a set(DO_COMMIT,...) fails, the core agent calls set(UNDO_COMMIT,handle,index) for each already committed triplet. The core agent then calls set(UNDO_PHASE1, handle, index) for all stored triplets in reverse order.

If all calls to set(DO_COMMIT, ...) were successfully performed, the core agentcalls set(DO_RELEASE, handle, index) for all stored triplets in reverse order toinsure orderly release of all locks and resources reserved.

2.1.5 Tick Function

The tick function is called between packets every 10-15 seconds in order to allow MIB servers to perform any time related functions: clean up old notInService rows, initiate trap sending, etc.

2.2 Program Flow

This section gives an overview of the program flows of a simple, straightforward conceptual core agent and a cooperating MIB server. For clarity and simplicity, this description omits error handling.

2.2.1 Initialization

Core Agent MIB server

```
for(all loadable MIB servers){
        add MIB server to MIB-server list;
        for(all MIB-server name spaces){
                for(all entities supported){
                    add mib to the list
                }
        }
}

for(all mibs known){
        mib->mib_specific = mib->mib_init();  ----->        reset to known state,
}                                                                   according to init
                                                                    string, remember
                                                            pointers to global
                                                                    data
```

2.2.2 GetRequest Processing

```
for(all mibs for particular entity){
        mark mib as not referenced;
}


for(each requested varbind)
        find first and last mib
        entries to search;
        for(
                mib=first;
                mib!=last;
                mib=mib->mib_next
        ){
                if(not referenced yet){
                        mib->mib_indicate(START)  ----->        remember pdu type, version,
                        mark as referenced;                     view and temporal domain,
                }                                       perform all operations
                                                                required to start packet
                                                        processing

                mib->mib_look()                 ------>  lok for variable instance,
        }                                                       check access, pass value
add result to output packet;                                    back
}

for(all mibs for particular entity){
        if(not referenced){
                continue;
        }
        mib->mib_indicate(END)                  ---->   perform all operations
}                                                       required to end packet
                                                        processing
build response packet and
return pointer to packet to the
communications layer
```

2.2.3 SetRequest Processing

```
for(all mibs for particular entity){
        mark mib as not referenced;
}


for(each requested varbind)
        find first and last mib
        entries to search;
        for(
                mib=first;
                mib!=last;
                mib=mib->mib_next
        ){
                if(not referenced yet){
                        mib->mib_indicate(START)   -->  remember pdu type, version,
```

```
                        mark as referenced;                     view, and temporal domain,
                }                                               perform all operations
        }                                                       required to start packet
                                                                processing

        mib->mib_look()                 -->     look for variable instance,
                                                check access, check value,
                                                        pass handle back
         if (handle != 0) {
                store mib, handle, index triplet
        }
}

for(all triplets){
        mib->mib_set(DO_PHASE1, handle, index) -->      check consistency, reserve
}                                                       resources

for(all triplets){
        mib->mib_set(DO_COMMIT, handle, index) -->      perform commit
}
for(all triplets in reverse order){
        mib->mib_set(DO_RELEASE, handle, index) -->     release resources
}

for(all mibs for particular entity){
        if(not referenced){
                continue;
        }
        mib->mib_indicate(END) -        ---->   perform all operations
}                                               required to end packet
                                                processing

build response packet and
return a packet pointer to the
communications layer
```

2.3 Message Types

For external MIB servers, there are corresponding request and response messages for each of the primitives listed above. The set of messages is defined in a later section.

3 ERROR CODES AND PSEUDO-TYPES

The Open Agent architecture defines a consistent set of intermediate error codes used throughout the agent. Below is the description of the error-codes applicable to the core-agent-to-MIB-server interface.

The Open Agent architecture uses an extended set of types internally. This set of types includes the base set of types defined by SMI and pseudo-types. The pseudo-type is a subtype of the base type which requires additional processing from the core agent.

There is just a single pseudo-type defined currently: WESA_TYPE_UINT32V1. There are some SNMPv1 mibs which use objects of type INTEGER (0...4294967295). The value of this type must be encoded in 5 bytes if it is greater than or equal to 0x80000000. WESA_TYPE_UINT32V1, defined as (0x100 | WESA_TYPE_INTEGER), gives the core agent the information required for appropriate encoding in this specific case.

Extended-type encoding is straightforward: the SMI type is encoded in byte 0 (LSB) of the extended type, and byte 1 is used to provide the core agent with additional information. If byte 1 is 0 then this is a basic SMI type, otherwise it is a pseudo-type. See section 4.2.3 for precise descriptions of where and how extended types are used.

If some undefined extended type is returned to the core agent, the agent generates genErr for delivery to the NMS. The core agent performs no conversions of SNMPv2 SMI types to SNMPv1 SMI types. It is the responsibility of the MIB server to use types in accordance with the version of the request packet being processed.

3.1 Intermediate Error Codes: Core-Agent-to-MIB-Server Interface.

WESA_ERR_INTERNAL-- the core agent will perform indicate(END) primitive for all namespaces already touched in the current packet processing and will drop processing of the request without any other action. This is meant as a hook for non-native proxies and multi-threaded implementations and should hardly ever be used in WinSNMP.

WESA_ERR_NOERROR -- same as SNMPv2 noError
WESA_ERR_GENERR -- same as SNMPv2 genErr
WESA_ERR_WRONGTYPE -- same as SNMPv2 wrongType
WESA_ERR_WRONGLENGTH -- same as SNMPv2 wrongLength
WESA_ERR_WRONGVALUE -- same as SNMPv2 wrongValue
WESA_ERR_NOCREATION -- same as SNMPv2 noCreation
WESA_ERR_NOACCESS -- same as SNMPv2 noAccess
WESA_ERR_INCONSISTENTVALUE -- same as SNMPv2 inconsistentValue
WESA_ERR_RESOURCEUNAVAILABLE -- same as SNMPv2 resourceUnvailable
WESA_ERR_NOTWRITABLE -- same as SNMPv2 notWritable
WESA_ERR_INCONSISTENTNAME -- same as SNMPv2 inconsistentName
WESA_ERR_NOSUCHOBJECT -- there is no object which can be matched to the requested name with max_access != non-accessible and either request type is Get, GetNext or Get-Bulk, or request version is SNMPv1 or requested name is mapped in by the current view.
WESA_ERR_NOSUCHINSTANCE -- there is no matching instance for a requested name and such an instance cannot be created in the case of a set operation and the requested/found name is mapped in by the current view
WESA_ERR_OBJECT_MAPPED_OUT -- there is no matching object with max_access != non-accessible for current request.

WESA_ERR_NAME_NOT_DETECTED -- this error is equivalent of genErr and is used by implementations with overlapped namespaces. TheMIB server must return this code when a generic error

occurred before the MIB server detected if the requested name is included in thecurrent namespeace.

3.2 Error Processing and Reporting

Any error not listed below for a particular case is an "unexpected error." For example, the error value 1002345 is an unexpected one for any case, and the error WESA_ERR_BADVALUE is an unexpected one for SNMPv1 GetRequest. Unexpected error codes returned for a particular environment are translated to genErr by the core agent. The next subsections of this section will describe error translation performed by the core agent for each environment.

In all cases, WESA_ERR_NAME_NOT_DETETCTED and WESA_ERR_INTERNAL are acceptable and expected error codes.

3.2.1 Look Primitive

3.2.1.1 SNMPv2 GetRequest

WESA_ERR_NOERROR	noError, core agent will create a varbind using returned type and value in the output packet
WESA_ERR_GENERR	genErr
WESA_ERR_NOSUCHOBJECT	
WESA_ERR_OBJECT_MAPPED_OUT	noError, core agent will create a varbind with value noSuchObject in the output packet
WESA_ERR_NOSUCHINSTANCE	
WESA_ERR_INSTANCE_MAPPED_OUT	noError, core agent will createa varbind with value noSuchInstancein the output packet

3.2.1.2 SNMPv2 GetNextRequest and GetBulkRequest

WESA_ERR_NOERROR	noError, core agent will createa varbind using returned type andvalue in the output packet
WESA_ERR_GENERR	genErr
WESA_ERR_NOSUCHOBJECT	
WESA_ERR_OBJECT_MAPPED_OUT	
WESA_ERR_NOSUCHINSTANCE	
WESA_ERR_INSTANCE_MAPPED_OUT	(a) If there are namespaces left unprocessedcore agent will continue search in thenext namespaces, otherwise(b) noError, core agent will createa varbind with value endOfMibViewin the output packet

3.2.1.3 SNMPv2 SetRequest

WESA_ERR_NOERRORr	noErro
WESA_ERR_GENERR	genErr
WESA_ERR_WRONGTYPE	wrongType
WESA_ERR_WRONGLENGTH	wrongLength
WESA_ERR_WRONGVALUE	wrongValue
WESA_ERR_NOCREATION	noCreation

WESA_ERR_INCONSISTENTVALUE inconsistentValue
WESA_ERR_RESOURCEUNAVAILABLE resourceUnvailable
WESA_ERR_NOTWRITABLE notWritable
WESA_ERR_INCONSISTENTNAME inconsistentName
WESA_ERR_NOSUCHOBJECT
WESA_ERR_NOSUCHINSTANCE
WESA_ERR_NOCREATION
noCreation
WESA_ERR_OBJECT_MAPPED_OUT
WESA_ERR_INSTANCE_MAPPED_OUT
WESA_ERR_NOACCESS
noAccess

3.2.1.4 SNMPv1 GetRequest and GetNextRequest

WESA_ERR_NOERROR noError, core agent will create a varbind using returned type and value in the output packet
WESA_ERR_GENERR genErr
WESA_ERR_NOSUCHOBJECT
WESA_ERR_OBJECT_MAPPED_OUT
WESA_ERR_NOSUCHINSTANCE
WESA_ERR_INSTANCE_MAPPED_OUT
noSuchName

3.2.1.5 SNMPv1 SetRequest
WESA_ERR_NOERROR noError
WESA_ERR_GENERR genErr
WESA_ERR_WRONGTYPE
WESA_ERR_WRONGLENGTH
WESA_ERR_WRONGVALUE
WESA_ERR_INCONSISTENTVALUE
badValue
WESA_ERR_RESOURCEUNAVAILABLE genErr
WESA_ERR_NOCREATION
WESA_ERR_NOTWRITABLE
WESA_ERR_INCONSISTENTNAME
WESA_ERR_NOSUCHOBJECT
WESA_ERR_NOSUCHINSTANCE
WESA_ERR_OBJECT_MAPPED_OUT
WESA_ERR_INSTANCE_MAPPED_OUT
WESA_ERR_NOACCESS
WESA_ERR_NOCREATION

noSuchName

3.3 Set Primitive

3.3.1 DO_PHASE1

The same set of rules is applicable here as in the case of the look primitive in the same environment.

3.3.2 UNDO_PHASE1, RELEASE

The return value and index value are ignored.

3.3.3 SNMPv1 CMD_DO_COMMIT

The same set of rules is applicable here as in the case of the look primitive in the SNMPv1 SetRequest environment.

3.4 SNMPv1 CMD_UNDO_COMMIT

The return value and index value are ignored.

3.5 SNMPv2 CMD_DO_COMMIT/CMD_UNDO_COMMIT

The same set of rules is applicable here as in case of the look primitive in the SNMPv2 SetRequest environment, if all CMD_DO_COMMIT calls return WESA_ERR_NOERROR. Otherwise the core agent stores the index passed back by the failed set(CMD_DO_COMMIT). If all subsequent CMD_UNDO_COMMIT calls return WESA_ERR_NOERROR, then the core agent generates commitFailed and error-index is set equal to the stored index value. Otherwise the core agent generates undoFailed with error-index 0.

4 FUNCTION-CALL INTERFACES FOR LOADABLE MIB SERVERS

This section describes the basic structures and primitives used in the core-agent-to-MIB-server interface when the MIB server is implemented as a loadable MIB server.

4.1 Basic Structures

4.1.1 The mib structure

The mib is the basic structure of the interface. To be accessible to the core agent, each mib namespace mounted under the agent must be represented by a single entry in a doubly-linked list of mib elements.

```
struct mib {
    struct mib FAR *mib_prev;
    struct mib FAR *mib_next;
    struct mib_server FAR *mib_server;
    INT32  mib_state;
    OID    mib_low_bound[MIB_BOUND_LEN];
    INT32  mib_low_bound_len;
    OID    mib_hi_bound[MIB_BOUND_LEN];
    INT32  mib_hi_bound_len;
    UINT32 mib_specific;
    UINT32 mib_priority;
    UINT32 (FAR PASCAL *mib_init)(const struct mib FAR *);
    BOOL   (FAR PASCAL * *mib_indicate)(INT32, UINT32, INT32,
                INT32, BOOL, INT32, INT32, void FAR *,
                const struct mib FAR *);
    INT32  (FAR PASCAL *mib_look)(LPOID, LPINT32, INT32, LPUINT8
                FAR *, LPUINT16, LPINT32, LPUINT32,
                const struct mib FAR *);
    INT32  (FAR PASCAL *mib_set)(INT32, UINT32, LPINT32,
                const struct mib FAR*);
    void   (FAR PASCAL *mib_tick)(UINT32, const struct mib FAR *);
    UINT8  mib_entity[MAX_ENTITY_LEN];
    INT32  mib_entity_len;
    INT32  mib_flag;
};
```

mib_prev and mib_next -- are pointers to the previous and next mib structures in the list. NULL means that this structure is the first or final one in the list.

mib_server -- is a pointer to the structure defining the MIB server which accesses the namespace that this mib structure defines.

mib_state -- describes the current state of the namespace represented by this mib structure. The core agent maintains one of several legal values in mib_state, including:

- namespace is loaded, but it's inactive and can't be used: mib functions can't be called.
- namespace is loaded and active: mib functions can be called.
- namespace is loaded but disabled by the core agent due to some malfunction discovered by the core agent: mib functions can't be called.

mib_low_bound -- is the low boundary of the namespace represented by and subsumed under this mib entry. The core agent looks at mib_low_bound to decide which active MIB server should be called to get

or set each variable in the current request. If name belongsto the namespace it must be lexicographically greater than or equal to the namespace's mib_low_bound.

mib_low_bound_len -- is the length of mib_low_bound.

mib_high_bound -- the high bound of the namespace represented by this mib entry. The core agent uses mib_high_bound and mib_low_bound to detect whether the namespace of a newly-mounted MIB server overlaps the namespace of any other MIB server. If name belongs to a namespace, then it must be lexicographically less than that namespace's mib_high_bound.

mib_high_bound_len -- the length of mib_high_bound.

mib_specific -- used by MIB-server code in some server--specific way. For, example, the value of mib_specific (returned by init(), stored by the core agent, and then made visible to the MIB server on each subsequent call) is the only way an external MIB server supporting several entities and/or namespaces can detect what entity and namespace is referenced in the current invocation. It is the responsibility of the MIB server to return some useful value from the mib_init() function which the core agent will then store into mib_specific.

mib_priority -- used by MIB servers handling overlapping namespaces. The highest priority is 0; a namespace with this priority cannot be overlapped with any other namespace. Normal priority is 1000.

mib_init -- pointer to the init primitive function in the MIB server for the namespace represented by this mib element. mib_init() returns 0 on failure, else some non-zero UINT32 value of interest only to the MIB server. The core agent stores this value into mib_specific.

mib_indicate -- pointer to the indicate primitive function in the MIB server for the namespace represented by this mib element. mib_indicate() returns FALSE on failure and TRUE on success.

mib_look -- pointer to the look primitive function in the MIB server for the namespace represented by this mib element. mib_look() returns an error code on failure.

mib_set -- pointer to the set primitive function in the MIB server for the namespace represented by this mib element.

mib_tick -- pointer to the tick primitive function in the MIB server for the namespace represented by this mib element.

mib_entity[] -- the local entity name.

mib_entity_len -- the length of mib_entity.

mib_flag -- used internally by the core agent.

4.2 Function Definitions

The formal definitions of all core-agent-to-MIB-server primitives are given below with reference to a fictitious example namespace called the "abc" namespace. The agent calls the functions by reference to a pointer in the MIB server's mib structure, so the "name" is that of the pointer, not the MIB server's actual function name. As we said above, the actual name of the function is a matter of choice for MIB server designer. We called them abc_xxxx just to logically link the primitive and namespace. The final parameter of each primitive function call is a FAR pointer to the mib entry itself which allows MIB-server access to its mib entry.

4.2.1 Init Primitive Function

Called by the core agent at the time the MIB server is mounted and before any other primitives are called.

Syntax:

UINT32 FAR PASCAL abc_init(struct mib FAR * mib);

struct mib FAR * mib -- pointer to the mib entry itself which allows MIB-server access to its mib entry.

Returns:

0 on failure and some UINT32 non-zero value on success. The core agent stores this returned value in mib_specific for later reference and use by the MIB server. If the MIB server returns zero, then this namespace is marked inactive and won't be referenced again. The return value is otherwise not interpreted in any way by the core agent.

Description:

The core agent issues init at the time the MIB server is mounted and before any other primitives are called.

4.2.2 Indicate Primitive Function

Called by the core agent for each namespace to indicate the start and end of processing for each received request packet.

Syntax:

```
BOOL FAR PASCAL abc_indicate(
  INT32   cmd,
  UINT32  cid,
  INT32   version,
  INT32   rw,
  BOOL    exact,
  INT32   view_ind,
  INT32   curr_time,
  void    FAR *view_head,
  const   struct mib FAR *mib
);
```

```
cmd             -- CYCLE_INDICATE_START or CYCLE_INDICATE_END
cid             -- cycle id
version         -- is either SNMPV1 or SNMPV2
rw        -- is either WESA_READ or  WESA_WRITE
exact           -- is either TRUE or FALSE
view_ind        -- is the index of the current view
curr_time       -- is either CURRENTTIME, RESTARTTIME or CACHETIME
view_head       -- is the pointer to the current view list.
```

Returns:

indicate returns FALSE on failure and TRUE on success. If indicate returns failure, then the core agent disables the MIB server, and all its mibs, and (if applicable) unload the MIB server.

If indicate(START) returns FALSE, then the core agent will respond to the NMS with error-status genError and error-index pointing to the first variable in the packet which has resides in the MIB server's namespace. If indicate(END) returns FALSE, this does not affect the result of current packet processing.But in both cases the core agent disables all the MIBs represented by this MIB server.

Description:

The core agent invokes the MIB server's indicate function for each namespace when a request packet is delivered from the communications layer, and before any other MIB-server functions are invoked for that namespace and packet; and when all the varbinds in that packet for each namespace have been processed and no further MIB-server functions will be called for that packet. The function intends to let each MIB server synchronize on the arrival of packets and the end of packet processing.

cmd takes the value CYCLE_INDICATE_START to signal the start of packet processing and CYCLE_INDICATE_END to signal the end of the current packet.

Recall that there may be several namespaces represented by a single MIB server and therefore several instantions of that MIB server in mib. What happens in this case? Imagine a MIB server which represents two namespaces and which therefore has two entries in mib: "abc_A" and "abc_B." Suppose further that the order of variables in a GetRequest pdu is as:

abc_A_1, abc_A_2, abc_B_1, abc_A_3, abc_B_2

Processing will be as:

```
abc_A_indicate(START)  -- server is told of a new packet
                          referencing the abc_A namespace.
abc_A_look(abc_A_1)    -- process GetRequest of abc_A_1;
                          abc_A_indicate(START) is guaranteed to
                          be invoked before abc_A_look().
abc_A_look(abc_A_2)    -- process GetRequest of abc_A_2.
abc_B_indicate(START)  -- core agent finds a reference to the
                          abc_B namespace, again tells server of
                          this packet.
abc_B_look(abc_B_1)    -- process GetRequest of abc_B_1; again,
                          abc_B_indicate(START) is guaranteed to
                          be invoked before abc_B_look().
abc_A_look(abc_A_3)
abc_B_look(abc_B_2)
abc_B_indicate(END)    -- the order of invocations to  abc_A_indicate(END)
                          abc_X_indicate(END) is unpredictable, but it is
                          guaranteed that there will be  no call to abc_X_look()
                          after any call  to abc_X_indicate(END) for any
                                namespace.
```

In order to distinguish the very first call (and the very beginning of the packet processing) cid is used. This 32-bit unsigned never-0 value which will never be 0, this value is incremented by the core agent for every packet processed. It wraps on overflow, skipping 0 in the process. The MIB-server can thus detect the very beginning of the packet. The same value is used by the core-agent to detect packets delivered after timeout.

rw and exact together tell the MIB server what request is present in this packet. If rw is WESA_READ and exact is TRUE, then the request is GET; if rw is WESA_READ and exact is FALSE, then the request is GET-NEXT or GET-BULK. if rw is WESA_WRITE, then the request is SET and exact is always TRUE and may be ignored.

4.2.3 Look Primitive Function

Invoked for all MIB-access operations.

Syntax:

```
INT32 FAR PASCAL abc_look(
  LPOID    name,
  LPINT32  namelen,
  INT32    index,
  LPUINT8  FAR *pval,
  LPUINT16 type,
  LPINT32  len,
  LPUINT32 ph,
  const struct mib FAR  *mib
);
```

Get Transaction:

```
name     -- requested name
namelen          -- pointer to its length
index    -- of current variable (ignored)
pval             -- *pval will point to unencoded value upon return
type     -- *type will contain extended type of variable found upon return
len              -- *len  will contain length of variable value upon return
ph               -- ignored
```

Next Transaction:

```
name     -- requested name upon return will contain the found name
namelen          -- pointer to its length
index    -- of current variable (ignored)
pval             -- *pval will point to unencoded value on return
type     -- *type will contain extended type of variable found
len              -- *len  will contain length of variable value
ph               -- ignored
```

Write Transaction:

```
name     -- name of variable
namelen          -- pointer to its length
index            -- of current variable, can be stored for following use
pval             -- *pval points to unencoded value.
type     -- type  points to the ASN type of value (*)
len              -- len   points to the length of value (**)
ph               -- *ph   will contain the non-zero handle upon return at MIB server's choice
```

Returns:

The value returned by look() is either WESA_ERR_NOERROR for success or some error code from Section 3. If this return value is an error code, then the core agent discards the other values passed back by the MIB server.

Description:

The look function is invoked for all MIB-access operations: GetRequest, GetNextRequest, GetBulkRequest, and SetRequest. The core agent communicates the type of transaction called out in the request packet in the invocation of indicate(), so the transaction type isn't repeated in the look() invocation.

The MIB server has to support instance-level granularity while calculating access rights for the particular name in all cases.

Notes:

(*) The SMI type is used here, not an extended type. It is the responsibility of the MIB-server designer to provide appropriate handling for the case if pseudo-type is used by variables of this MIB-server.

(**) If *len == -1 then some error was detected during value decoding.

4.2.4 Set Primitive Function

Used to set variables.

Syntax:

```
INT32 FAR PASCAL abc_set(
   INT32   cmd,
   UINT32  hand,
   LPINT32 pind,
   const struct mib FAR *mib
);
 cmd    -- can take five values: CMD_DO_PHASE1, CMD_UNDO_PHASE1,
             CMD_DO_COMMIT, CMD_UNDO_COMMIT, CMD_DO_RELEASE

handle  -- non-zero handle returned by MIB server's look primitive

index   -- points to the variable for which the MIB server returned a non-zero.
```

Returns:

Return values of this function are described in the section 3.

Description:

Described in detail in section 3.

4.2.5 Tick Primitive Function

Informs the MIB server of the passage of time.

Syntax:
void FAR PASCAL party_tick(UINT32 cid, const struct mib FAR *mib)

cid -- holds a non-zero 32-bit value incremented by one before each functioninvocation.
Returns:

Nothing.

Description:

The core agent invokes the tick function once every few (nominally ten) seconds between packets to allow the MIB server to perform any needed time-related functions. The tick function is never called while any packet is being processed, but only between request packets.

5 MESSAGE-PASSING INTERFACE FOR EXTERNAL MIB SERVERS

This section describes the invocation of primitives as it's realized for external MIB servers. The message-passing mechanism between a WinSNMP core agent and external MIB servers uses a combination of native MS-Windows messaging and shared memory.

5.1 Combined Message/Shared-Memory Mechanism

All WinSNMP agent message exchanged between the WinSNMP Agent and its external MIB servers have the following format:

```
struct defWinSNMPMsg {
    WHND     hWnd;            /* handle of receiving window   */
    WORD     wMessageType;    /* RegisterWindowMessage(
                                  "WinSNMPAgentMessageType")   */
    WORD     wParam;          /* WESA_<message sub-type>      */
    LONG     lParam;          /* HIWORD=shared memory handle  */
                              /* LOWORD=sending window handle */
};
```

5.1.1 WinSNMP Agent Windows Message Type

There is one type for all WinSNMP Agent messages exchanged between the core agent and external MIB servers via PostMessage and SendMessage. This messages type is determined dynamically on each system by issuing:

```
WORD wMessageType =
RegisterWindowMessage("WinSNMPAgentMessageType");
```

5.1.2 WinSNMP Agent Message Sub-Type

There is a constant message sub-type for each interface primitive's message and its correspondent response:

Primitive	Primitive Invocation	Primitive Response
init()	WESA_INIT_REQ	WESA_INIT_RSP
indicate()	WESA_INDICATE_REQ	WESA_INDICATE_RSP
look()	WESA_LOOK_REQ	WESA_LOOK_RSP
set()	WESA_SET_REQ	WESA_SET_RSP
tick()	WESA_TICK_REQ	WESA_TICK_RSP

The message sub-type is passed in the wParam of all WinSNMP Agent windows messages.

5.1.3. Shared Memory Areas

There are three shared-memory areas involved in exchanges of information between the WinSNMP core agent and its correspondent external MIB Servers, known as Areas
B, C and D:

- Area B is used to store the list of views known to the system. The actual layout of this area is Agent Core implementation specific and is not seen by the MIB servers.
- Area C is used by the core agent as a link area for core-agent-to-MIB-server transactions.
- Area D is used by the MIB server as a link area for MIB-server-to-core-agent transactions.

The MIB-server never writes in areas B and C; the MIB server writes only into Area D. The core-agent never writes into Area D but only into Areas B and C.

The shared-memory areas are allocated from Global Heap and are assumed to be moveable. The handle to the shared-memory area is passed in the high order word of lParam in WinSNMP Agent windows messages.

5.1.4 Sending Window's Handle

Each message identifies the sending window's handle. This handle issued to identify the message source for response destination and message context selection. The handle is passed in the low-order word of the lParam in WinSNMP Agent windows messages.

5.1.5 Message Id

Each message sent in either direction (to the MIB server by the core agent; or to the core agent by the MIB server) must contain an individual Message ID. Message ID is the cycle identifier (cid from indicate() primitive description above). For all other messages it is a non-zero unsigned 32-bit value assigned by the core agent. The message ID is passed in the shared memory associated with message.

5.2 Message and Data Structures

The messaging mechanism in this section is a direct mapping of the functional interface described above, so this section will not repeat any general information about the primitives themselves.

In the diagrams below, we place the core agent on the left side of the page and the MIB server on the right.

```
Agent            <Message>                    MIB Server
====                                          =========
------------------<message sent by agent>------------------>
<-----------------<message sent by MIB Server>--------------
```

The full message format is

```
<hWnd><msg_type><msg_subtype><shared_area_hdl | sending_window_hdl>
                          (wParam)        (high lParam)       (low lParam)
```

In the abbreviated diagrammatic message format used, only the wParam as <message_subtype> and the high order lParam as <shared_area_handle> appear as follows:

```
<message_subtype><shared_area_handle>
 (wParam)               (high lParam)
```

5.2.1 Init

Message Exchange:

```
--------------<WESA_INIT_REQ><Handle of shared area C>----------->
<--------------<WESA_INIT_RSP><Handle of shared area D>--------------
```

Area C format :

Offset	Type	Contents
0	UINT32	Message ID
4	UINT32	Length of the Entity Name (0..32)

8	UINT8[]	EntityName
40	UINT32	Length of namespace's low boundary (in OIDs)(2..20)
44	OID[]	Low boundary
124	UNIT32	Handle of shared area B (obsolete, will be removed)
128	UINT32	Current value of sysUpTime (obsolete, will be removed)

Area D format:

Offset	Type	Contents
0	UINT32	Message ID
4	UINT32	Result of operation (will be stored in mib_specific, see 4.2.1 above)

5.2.2 Indicate

Message Exchange:

----------<WESA_INDICATE_REQ><Handle of shared area C>---------->
<----------<WESA_INDICATE_RSP><Handle of shared area D>----------

Area C format :

Offset	Type	Contents
0	UINT32	cycle-id
4	UINT32	mib_specific
8	UINT32	cmd
12	UINT32	version
16	UINT32	rw
20	UINT32	exact
24	UINT32	current view index
28	UINTY32	current time domain
32	UINT32	handle of shared area B (view list)

Area D format:

Offset	Type	Contents
0	UINT32	cycle-id
4	UINT32	Result of operation

5.2.3. Look

a) get transaction

Message Exchange:

----------<WESA_LOOK_REQ><Handle of shared area C>---------->
<----------<WESA_LOOK_RSP><Handle of shared area D>----------

Area C format :

Offset	Type	Contents
0	UINT32	message-id
4	UINT32	mib_specific
8	UINT32	name length (in OIDs)
12	OID[]	name

Area D format:

Offset	Type	Contents
0	UINT32	message-id
4	UINT32	result of operation
8	UINT32	value type
12	UINT32	value length (unencoded)
16	any	value (unencoded)

b) next transaction

Message Exchange:

----------<WESA_LOOK_REQ><Handle of shared area C>---------->
<----------<WESA_LOOK_RSP><Handle of shared area D>----------

Area C format :

Offset	Type	Contents
0	UINT32	message -id
4	UINT32	mib_specific
8	UINT32	name length (in OIDs)
12	OID[]	name

Area D format:

Offset	Type	Contents
0	UINT32	message-id
4	UINT32	result of operation
8	UINT32	name length (in OIDs)
12	OID[]	name
524	UINT32	value type
528	UINT32	value length (unencoded)
532	any	value (unencoded)

c) write transaction

Message Exchange:

----------<WESA_LOOK_REQ><Handle of shared area C>---------->
<----------<WESA_LOOK_RSP><Handle of shared area D>----------

Area C format:

Offset	Type	Contents
0	UINT32	message-id
4	UINT32	mib_specific
8	UINT32	name length (in OIDs)
12	OID[]	name
524	UINT32	variable index

528	UINT32	value type
532	UINT32	value length (unencoded)
536	any	value (unencoded)

Area D format:

Offset	Type	Contents
0	UINT32	message-id
4	UINT32	result of operation
8	UINT32	handle or 0

Note: the core agent communicates the type of the transaction specified in the request packet in the indicate(START) message so the transaction type isn't repeated in the look() invocation.

5.2.4. Set

Message Exchange:

----------<WESA_SET_REQ><Handle of shared area C>---------->
<----------<WESA_SET_RSP><Handle of shared area D>----------

Area C format:

Offset	Type	Contents
0	UINT32	message-id
4	UINT32	mib_specific
8	UINT32	cmd
12	UINT32	handle
16	UINT32	variable index

Area D format:

Offset	Type	Contents
0	UINT32	message-id
4	UINT32	result of operation
8	UINT32	variable index

5.2.5 Tick

Message Exchange:

----------<WESA_TICK_REQ><Handle of shared area C>---------->
<----------<WESA_TICK_RSP><Handle of shared area D>----------

Area C format:

Offset	Type	Contents
0	UINT32	tid
4	UINT32	mib_specific
8	UINT32	sysUpTime

Area D format:

Offset	Type	Contents
0	UINT32	tid

6 PROCEDURES FOR MOUNTING AND UNMOUNTING OF MIB SERVERS

6.1 Loadable MIB Servers

Each loadable MIB server defines three functions:

```
BOOL  FAR PASCAL  server_reg_init(char FAR *init_str);
INT32 FAR PASCAL  server_reg_next(struct mib FAR *mib);
void  FAR PASCAL  server_unreg(INT32 reason);
```

The function server_reg_init() performs all the operations needed for the MIB server to mount itself.

init_str is initialization data for use at MIB-server mounting time and which depends on the needs of the MIB server. The value of init_str is retrieved by the core agent from the configuration database, and this value is passed to this function without any further processing. This value can be the name of MIB server private configuration file or anything else. Note: init_str is not used by current implementation, it will be apparently gone very soon. Current implementation assumes that each loadable server is smart enough to read its own .ini file without any help.

The server_reg_init() function returns FALSE on failure else TRUE.

Upon successful invocation of server_reg_init(), the core agent allocates a new mib structure entry and passes a pointer to it as a parameter to function server_reg_next(). That function stores appropriate values in the following members of the pointed mib_list entry: mib_low_bound, mib_high_bound, mib_low_bound_len, mib_high_bound_len, mib_priority, mib_init, mib_indicate, mib_look, mib_set, mib_view, mib_tick, mib_entity, and mib_entity_len.

There are three legal return values from the function:

- MSR_SUCC on success, in which case the function is called again with a pointer to another newly-assigned mib_list entry;
- MSR_NOMORE if there are no further namespaces to register;
- MSR_ERROR on failure.

The function server_unreg causes the MIB server to perform any operations needed to terminate its activities. The reason parameter can take three value:

- MSU_NOMOUNT, meaning there is some overlapping namespace;
- MSU_COMMAND, meaning a command to unmount the MIB server was received;
- MSU_OTHER, meaning any other case.

6.2 External MIB Servers

There are two ways the external MIB server can be brought to life. It can be started by the core agent or it can be started by the user as part of any application. The same handshake procedure is used in either case.

6.2.1 External MIB-Server Descriptor

Every external MIB server must initialize its namespaces and allocate global shared memory area using its Server Descriptor. The external MIB server needs to maintain its descriptor in a consistent state while it is running. The external MIB server change its descriptor, but these changes have to be done atomically. The Server Descriptor has this format:

Server Descriptor format:

Offset	Type	Contents
0	char	Unique name of MIB server, null terminated
80	char	Description of MIB server, null terminated
160	UINT32	Number of namespaces this MIB server will handle
First namespace handled by this MIB server		
164	UINT32	mib_specific
168	UINT32	mib_priority
172	UINT32	Length of entity name
176	UINT8[]	Entity name
208	UINT32	Length of low boundary in OIDs
212	OID[]	Low boundary
292	UINT32	Length of high bound in OIDs
296	OID[]	High boundary
Second namespace handled by this MIB server		
376	UINT32	mib_specific
Information for second and subsequent namespaces arranged as for the first		

6.2.2 MIB-Server Handshake

The MIB Server initiates mounting upon startup and after any changes to the MIB-Server descriptor.To do so it broadcasts the following message:

```
SendMessage(
   HWND_BROADCAST,
   wesa_msg_type,
   WESA_MIB_REG_REQ,
   MAKELONG(extMIBServerWindow, extMIBServerDescriptor)
);
```

The lParam of this message contains the MIB server's window handle in the LOWORD and the serverdescriptor handle in HIWORD. If there is a core agent running in the system it will respondby a sending message back to the server (while processing the message received fthe rom server, similarto DDE handshake):

```
SendMessage(
   extMIBServerWindow,
   wesa_msg_type,
   WESA_MIB_REG_RSP,
   MAKELONG(coreAgentWindow, 0)
);
```

The LOWORD of lParam in this message contains the core agent's window handle.

If the MIB server terminates, it posts a message to the core agent:

```
PostMessage(
   coreAgentWindow,
   wesa_msg_type,
   WESA_MIB_REG_ABORT,
   MAKELONG(extMibServerWindow, 0)
);
```

The LOWORD of lParam in this message contains the MIB server's window handle.

6.2.3 Core Agent Handshake

The core agent has to broadcasts the following message upon start up:

```
SendMessage(
    HWND_BROADCAST,
    wesa_msg_type,
    WESA_AGENT_REG_REQ,
    MAKELONG(coreAgentWindow, 0)
);
```

The LOWORD of lParam in this message contains the core agent's window handle.

Each external MIB server running in the system it responds by sending a message back to the core-agent (while processing the message received from the core-agent, similar to DDE handshake):

```
SendMessage(
    coreAgentWindow,
    wesa_msg_type,
    WESA_AGENT_REG_RSP,
    MAKELONG(extMIBServerWindow, extMIBServerDescriptor)
);
```

The lParam of this message contains the server's window handle in the LOWORD and the server descriptor handle in HIWORD.

If the core agent is unable to mount a MIB server; or wishes to unmount a MIB-server for some reason, it either sends or posts the following message:

```
SendMessage(
    extMIBServerWindow,
    wesa_msg_type,
    WESA_AGENT_REG_ABORT,
    MAKELONG(coreAgentWindow, reason)
);

PostMessage(
    extMIBServerWindow,
    wesa_msg_type,
    WESA_AGENT_REG_ABORT,
    MAKELONG(coreAgentWindow, reason)
);
```

The LOWORD of lParam in this message contains the core agent's window handle; the HIWORD of lParam in this message contains the reason for unmounting, where reason can be:

- MSU_NOMOUNT, meaning there is some overlapping namespace;
- MSU_COMMAND, meaning a command to unmount the MIB server was received;
- MSU_OTHER, meaning any other case.

6.3 Core Agent Mutual Exclusion

Upon startup, the core agent must check that there are no other WESA agents running in the system. To do so it broadcasts the following message:

```
SendMessage(
    HWND_BROADCAST,
    wesa_msg_type,
```

```
    WESA_AGENT_EXCL_REQ,
    MAKELONG(coreAgent1WindowHandle, 0)
);
```

If there is such an agent running in the system, it sends a message back (while processing this message, in DDE handshake style):

```
SendMessage(
    coreAgent1WindowHandle,
    wesa_msg_type,
    WESA_AGENT_EXCL_RSP,
    MAKELONG(coreAgent2WindowHandle, 0)
);
```

If the core agent receives the message WESA_AGENT_EXCL_RSP it must terminate.

7 DECLARATIONS

<to be added>

8 OTHER ISSUES

This section describes the execution environment of MIB servers and the integration of WinSNMP/Agent with DMTF.

8.1 Common services

There are several common services provided with the core agent, outlined below.

8.1.1 OID Comparison

The compare() function compares two oids.

```
INT32 FAR PASCAL compare(
    LPOID name1,
    INT32 len1,
    LPOID name2,
    INT32 len2
);
```

The function returns 1 if name1 is lexically greater than name2; returns 0 if names are the same; and returns -1 otherwise.

8.1.2 Views

```
INT32 FAR PASCAL check_view(
   LPOID  name,
   INT32  known_len,
   INT32  total_len,
   void   FAR *view_list,
   INT32  view_ind
);
```

This function evaluates whether a name is mapped into or out of the view represented by view_ind and view_list. Name is the name in question, total_len is the total length of the name, known_len is the length

of the part of the name which is known currently. For example, total_len can be the length of the variable name including the index part; and known_len can be the length of the object part of the name. View_list is a pointer to the list of view entries to be used. View_index is the index of the view of interest. This function returns CHV_YES if the name is mapped into the view; CHV_NO if the name is mapped out of the view; and CHV_MAYBE if there is not enough information to detect whether it is mapped in or out.

There is a variant of this function intended for use by external MIB servers.

```
INT32 FAR PASCAL check_view_sh(
   LPOID   name,
   INT32   known_len,
   INT32   total_len,
   HGLOBAL view_list_h,
   INT32   view_ind
);
```

The only difference between these two functions is that, in the second case, the handle of the view list head
is used instead of a pointer to the view head.

8.1.3 RowStatus State Machine

There is also a RowStatus state machine provided by the core agent.

```
INT32 FAR PASCAL _export rs_machine(
   BOOL     new_entry,
   LPINT32  new_status,
   INT32    status_index,
   INT32    old_status,
   BOOL     consistent,
   INT32    cons_index,
   LPINT32  pind,
   LPUINT32 tmout
);
```

This function returns an intermediate error code for the current operation in accord with the RowStatus transition table from RFC 1443.

New_entry is FALSE if the current operation is a modification of an existing entry, otherwise it is TRUE. If the status element is provided within the current packet, then *new_status is equal to this value on input; otherwise *new_status is equal to the old value of the status element on input. *new_status is the new value of the status element on output, which can be different from the value provided in the packet (CreateAndGo -> active) or different from the old value (notInService -> notReady). If the status element is provided within the current packet, then status_index is its index in the packet, otherwise it is 0. Old_status is the old value of the status element if this is a modification of an existing row, otherwise ignored. Consistent is TRUE if the rest of the new row's values (except status element) are consistent, otherwise it is FALSE. If consistent is FALSE and the inconsistency is related to any of the variables (except status element) in the current packet cons_index is the index of this variable otherwise it is 0. *pind is the value of the index of the first row variable in the current packet and, on output, *pind is the value of error-index. *tmout contains the time when this row should be discarded due to time-out in notInService or notReady status.

8.1.4 Time Service

There is a time service provided by agent, which give both UNIX-like time and sysUpTime values available to every MIB server.

```
UINT32 FAR PASCAL wesa_time(void)

UINT32 FAR PASCAL wesa_up_time(void)
```

8.1.5 Traps

There are two key elements in trap sending: the trap_register function and the trap data block.Any component of the WESA subsystem can send a trap by calling the trap_register function:

```
BOOL FAR PASCAL trap_register(
   INT32  tclass,
   UINT32 trap,
   UINT32 specific,
   UINT32 data
);
```

Data for non-generic (non-well known) traps are provided in the form of a trap data block.
This block is allocated from shared memory by the element sending the trap.
Shared memory is freed by the caller once the trap is sent. The trap data block has the following format:

Offset	Type	Value	Description
0	UINT32	trap_oid_len	length of next field
4	OID[]	trap_oid	SNMPv1: enterprise, SNMPv2: trap name
4+trap_oid_len*4	UINT32	trap_var_num	number of trap vars
8+trap_oid_len*4	struct trap_var[]	trap_vars	arry of trap vars

Where struct trap_var is defined as :

```
#define MAX_TRAP_VAR_OID 32   /* should be OK for next 15 years */
#define MAX_TRAP_VAL_LEN 256  /* more than enough */
struct trap_var {
   OID     tv_name[MAX_TRAP_VAR_OID];
   INT32   tv_name_len;
   UINT8   tv_val[MAX_TRAP_VAL_LEN]; /* value */
   INT32   tv_len;               /* tv_val length, non encoded */
   UINT16  tv_type;              /* pseudo-type */
};
```

8.1.5.1 SNMPv1 Generic Traps

In order to send a generic SNMPv1 trap, the following values must be passed to thefunction trap_register:

```
tclass     - TCLASS_V1_GENERIC
trap       - generic trap value
specific   - ignored
data       - if trap is linkUp or linkDown, data is ifIndex of link
             if trap is egpNeighborLoss, data is IP address of  neighbor
             in network byte order
```

8.1.5.2 SNMPv1 Specific Traps

In order to send a specific SNMPv1 trap, the following values must be passed to thefunction trap_register:

```
tclass     - TCLASS_V1_SPECIFIC
```

trap - generic trap value
specific - specific trap value
data - handle of trap data block

8.1.5.3 SNMPv2 Well-Known Traps

In order to send a well-known SNMPv2 trap, the following values must be passed to thefunction trap_register:

tclass - TCLASS_V2_WELL_KNOWN
trap - last oid of well-known trap name
specific - ignored
data - if trap is linkUp or linkDown, data is ifIndex of link
if trap is egpNeighborLoss, data is IP address of neighbor
in network byte order

8.1.5.4 SNMPv2 Non-Well-Known Traps

In order to send any SNMPv2 trap the following values must be passed to thefunction trap_register:

tclass - TCLASS_V1_GENERAL
trap - ignored
specific - specific trap value
data - handle of trap data block

The trap sending routine will make sysUpTime the first variable in the varbind,the snmpTrapOID will be the second one, and the variables from trap data block follow these two

8.2 Interface to DMTF

This section describes the proposed interface between the WinSNMP/Agent and DMTF. Because a DLL implementation does not offer the capabilities that DMTF requires, DMTF must be implemented as an external MIB server. This external MIB server will assume the role of a DMTF management application. On startup, the DMTF MIB server does the following:

- Register itself as a management application with the DMI Service Layer.
- Issue several list commands to obtain information about availability of DMTF managed entities; build appropriate transaction tables (or functions) to translate SNMP names into DMI names.
- Build appropriate namespaces representing discovered DMTF variables, build a registration data block, and register itself with core agent by sending WM_SNMP_IAMH_REQ message.

The mapping of WinSNMP/Agent primitives onto DMI primitives is straightforward. There is nothing DMTF specific which has to be done by a MIB server in order to perform init(), indicate(), and tick() primitives.

The mapping of the get and next transactions of the look() primitive is clear: perform name translation and call the appropriate DMTF list function(s). Then call the appropriate DMTF get-attribute function(s), store value in the static area and pass the pointer to this value, type, and length to the core agent.

The mapping of the write transaction is as:
Check the new value.
Perform name translation and call the appropriate DMTF list function(s), then call the appropriate DMTF get-attribute function(s).
Store the old and new values in a shadow area and pass the non-zero handle (if needed) to the core agent.

The function set(DO_PHASE1) is directly mapped onto the DMTF reserve commands. The function

set(UNDO_PHASE1) is directly mapped onto the DMTF release commands. The function set(DO_COMMIT) is directly mapped onto the DMTF set commands. The function set(UNDO_COMMIT) is mapped onto the DMTF set commads restoring the old values.

All non-solicited events indicated to the DMTF MIB server are translated by the MIB server into WM_SNMP_TRAP1_REQ and WM_SNMP_TRAP2_REQ messages.

9 References

9.1 Requests for Comment

1155 M. Rose and K. McCloghrie: Structure and Identification of Management Information for TCP/IP-based Internets, May 1990.

1157 J. Case, M. Fedor, M. Schoffstall, and C. Davin: Simple Network Management Protocol (SNMP), May 1990.

1213 K. McCloghrie and M. Rose: Management Information Base for Network Management of TCP/IP-based Internets: MIB-II, Mar 1991.

1215 M. Rose: Convention for defining traps for use with the SNMP, Mar 1991.

1441 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser:
Introduction to version 2 of the Internet-standard Network
Management Framework, Apr 1993.

1442 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser: Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2), Apr 1993.

1443 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser: Textual Conventions for version 2 of the Simple Network Management Protocol (SNMPv2), Apr 1993.

1444 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser: Conformance Statements for version 2 of the Simple Network Management Protocol (SNMPv2), Apr 1993.

1445 J. Galvin and K. McCloghrie: Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2), Apr 1993.

1446 J. Galvin and K. McCloghrie: Security Protocols for version 2 of the Simple Network Management Protocol (SNMPv2), Apr 1993.

1447 K. McCloghrie and J. Galvin: Party MIB for version 2 of the Simple Network Managment Protocol (SNMPv2), Apr 1993.

1448 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser: Protocol Operations for version 2 of the Simple Network Managment Protocol (SNMPv2), Apr 1993.

1449 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser: Transport Mappings for version 2 of the Simple Network Managment Protocol (SNMPv2), Apr 1993.

1450 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser: Management Information Base for version 2 of the Simple Network Managment Protocol (SNMPv2), Apr 1993.

1451 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser: Manager-to-Manager Management Information Base, Apr 1993.

1452 J. Case, K. McCloghrie, M. Rose, and S. Waldbusser: Coexistence between version 1 and version 2 of the Internet-standard Network Management Framework, Apr 1993.

9.2 Other Reference Sources

B. Natale: Windows SNMP: An Open Interface for Programming Network Management Applications using the Simple Network Management Protocol under Microsoft Windows, v1.0, Sep 13 1993.

S. Pendse: WinSNMP/MIB: An Interface for Programming using the Management Information Base of SNMP under Microsoft Windows, v1.0d, Oct 1993.

D. Perkins: Understanding SNMP MIBs, Revision 1.1.5, Jul 7 1992.

M. Rose: The Simple Book: An Introduction to Management of TCP/IP-base Internets, Prentice-Hall, 1990.

M. Rose: The Simple Book: An Introduction to Internet Management,Prentice-Hall, 1994.

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

APPENDIX A MIB-SERVER CODE EXAMPLE

<to be added>