



## Creating Network Element Modules

---

This chapter describes how to create a network element module. If the network element that you want to integrate with BBSM is compatible with BBSM, you can use the generic network element DLL instead of creating a new module. In this case, you only need to add a new registry entry to indicate that GenericSwitch.dll supports this type of network element.

To determine if this network element module supports the SNMP managed objects that BBSM requires, perform Cisco’s qualification procedure. Refer to [“Qualifying the Network Element for BBSM” section on page 7-3](#). This section also describes how to add the needed registry entry if you do not need to create a new DLL.

If your network element does not function as a “generic switch,” your subclass must override some methods of the CEtherStack base class. The SDK provides a class, CSNMPAgent, to facilitate generating SNMP requests to your network element, which assumes that you can determine the port and connection status through queries to a private MIB implemented by the SNMP agent on your network element. You are not limited to using SNMP to query your network element. Any mechanism that returns the information that the overridden method requires is acceptable.

This chapter provides the following:

- [Network Element Overview, page 7-1](#)
- [Qualifying the Network Element for BBSM, page 7-3](#)
- [Registry Entries for Network Element Modules, page 7-13](#)
- [Creating Your Network Element Module, page 7-6](#)
- [Network Element Module API Reference, page 7-16](#)

## Network Element Overview

Network elements provide connectivity between client computers and the BBSM internal NIC. Even though the term “switch” is often used to refer to a network element, BBSM supports not only Ethernet switches, but also the following other network elements:

- CTMSs (cable modems)
- Access points
- Digital subscriber line access multiplexers (DSLAMs)
- DSL-like voice over data solutions; for example, LRE

A BBSM network element module provides the interface between BBSM core functionality and the network elements that clients connect to. These modules provide a standard interface for the core to do the following:

- Populate a map of client ports that the network element provides
- Query the element to find where a client is located
- Determine whether the client is still connected to the BBSM network

The BBSM administrator populates and configures the port settings through the BBSM Dashboard. These calls are made from the context of the IIS web interface.

Queries for locating the client come from core BBSM ASP files invoked when the end user first attempts to access the Internet through BBSM. With this information, BBSM looks up the port map entry to determine the page set to display to the end user. This page set invokes the access and accounting policies configured for that port. These calls are also made in the context of IIS.

Finally, when an end user has authenticated through BBSM, the AtDial service monitors the status of the end user. When the end user turns off the client or physically disconnects from the network element, the service detects this event by sending a query to the network element through the network element module. This call occurs in the context of an NT service.

For BBSM to perform session management for a client connected to a network element, it must be able to determine two things:

- The port on the network element that the client computer connects to
- The duration of a client's BBSM session.

The BBSM software architecture allows third parties to add software modules, traditionally called "switch DLLs" but more properly called network element modules, to query network elements to obtain the information required by the BBSM.

BBSM is implemented on the Windows 2000 platform, and network element modules are MFC extension DLLs. These DLLs export both entry points and C++ classes. To add support for a new network element module, you do the following:

- Create an MFC extension DLL that exports a subclass of the CEtherStack class implemented in GenericSwitch.dll, which is supplied with the SDK.
- Add entries to the Windows 2000 registry to register your network element module with BBSM.

The CEtherStack class implements support for network elements that function as transparent bridges and implements the SNMP MIBs described in the following RFCs:

- RFC 1213: *Management Information Base for Network Management of TCP/IP-Base Internets: MIB-II*
- RFC 1493: *Definitions of Managed Objects for Bridges*

The CEtherStack class issues SNMP queries to search the forwarding database of the bridge for the port corresponding to the client computer MAC address and searches the Interfaces table to determine the connection status of the link.

## Qualifying the Network Element for BBSM

This section describes how to determine if a network element implements the SNMP-managed objects required to allow BBSM to interface with it by using the CEtherStack base class. The required objects are part of the MIBs defined in RFCs 1213 and 1493. This section also describes how to add the needed registry entry if you do not need to create a new DLL.

BBSM can support many Ethernet switches, especially the most recent versions, with the GenericSwitch.dll provided in BBSM. This DLL requires a correct implementation of the SNMP-managed objects listed in [Table 7-1](#). Test your network element with the following qualification procedure, because experience has shown that implementations of these managed objects sometimes have subtle errors that require workarounds.

**Table 7-1** SNMP-Managed Objects

MIB Object	Specified in	Description
dot1dTpAgingTime	RFC 1493	How long entries remain in the table (R/W)
dot1dTpFdbPort		Contains the port number, indexed by MAC address
dot1dTpFdbStatus		Contains the entry status (valid/invalid)
dot1dBasePortIfIndex		Maps port numbers to interface indices
IfOperStatus	RFC 1213	Should report UP only if a device is connected to the port
SysObjectID		Identifies the switch type

The entries specified in RFC 1493, “Definitions of Managed Objects for Bridges,” are found in the forwarding databases of the devices. The forwarding database of a transparent bridge tells the unit the port on which to transmit received packets based on the destination MAC address. We use this information to locate the client. The remaining objects are found in RFC 1213, “Management Information Base for Network Management of TCP/IP-based Internets: MIB-II” (<http://ietf.org/rfc.html>).

For the devices that support the MIB objects in [Table 7-1](#), use the following template to create a registry entry. Replace the items in **bold type**.

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IPOINT\Switches\
  YourSwitch]
"Code"=dword:YourSwitchNumber (hex)
"CreateStack"="CreateGeneric" "Description"="Your switch description"
"DLLPath"="c:\atcom\install\genericswitch.dll"
"PortCount"=dword:YourSwitchPortCount (hex)
"PortMapOnTheFly" = Your port map on the fly flag
"PortType" = Your port type
"SysObjectID"="Your switch sysObjectID"
"SwitchCapability" = Your cluster
"SwitchModeList"=dword:YourSwitchModeList
```



### Note

For a repository of all RFCs, refer to the Internet Engineering Task Force site at the following URL: <http://www.ietf.org/rfc.html>.

The following is required to test the network element:

- A BBSM server
- The network element to be tested
- A client
- The appropriate cabling

The BBSM server ships with a utility, `SNMPutil`, used to send SNMP requests and display the responses. Before performing the test, use `SNMPutil` to probe the network element's MIB. Using the `SNMPutil` procedure instead of the operational BBSM software makes it easier to determine whether or not the network element conforms to the specifications and, if the network element does not conform, how it fails to conform. For the examples that follow, these values were used:

```
Client IP = 192.168.3.130
Client MAC = 00-60-08-DE-C1-D2
Network Element IP = 192.168.3.221
Network Element SNMP community string = public
BBSM internal NIC = 192.168.3.1
```

To test the network element's SNMP implementation:

- 
- Step 1** Connect the network element to the BBSM server's internal network and a client to the network element. Run `ipconfig/all` or `winipcfg.exe` on the client to determine the MAC address of the client's NIC. See the following example.

Ethernet adapter El90x2:

```
Description . . . . . : ELPC3R Ethernet Adapter
Physical Address . . . . . : 00-50-DA-17-D2-3D
DHCP Enabled . . . . . : Yes
IP Address . . . . . : 192.168.3.130
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.3.1
Primary WINS Server . . . . . : 192.168.3.1
```

- Step 2** At the BBSM server, walk the `dot1dTpFdbAddress (17.4.3.1.1)` entry of the bridge forwarding database table on the network element.




---

**Note** Values represented in **bold** vary based on the developer's system.

---

```
C:\> snmputil walk 192.168.3.221 public 17.4.3.1.1
Variable = .17.4.3.1.1.0.80.218.23.209.105
Value = OCTET STRING - <0x00><0x50><0xda><0x17><0xd1><0x69>

Variable = .17.4.3.1.1.0.96.8.222.193.210
Value = OCTET STRING - <0x00><0x60><0x08><0xde><0xc1><0xd2>

End of MIB sub tree.
```

- Step 3** The second entry matches the client MAC address. Use the last 6 octets of the object ID (variable) to get the specific `dot1dTpFdbPort (17.4.3.1.2)` object for the network element. Use a "get" to perform this step, but if this does not work, try using a "walk." In the following, you can see that the client is connected to port 2 on the network element at IP address 192.168.3.221.

```
C:\>snmputil get 192.168.3.221 public 17.4.3.1.2.0.96.8.222.193.210
Variable = .17.4.3.1.2.0.96.8.222.193.210
Value    = INTEGER - 2
```

- Step 4** Verify that the dot1dTpFdbStatus (17.4.3.1.3) variable is implemented correctly. Entries in the table may be expired, etc., and we need to determine whether or not they are valid. The return value should be 3 (Learned):

```
C:\>snmputil get 192.168.3.221 public 17.4.3.1.3.0.96.8.222.193.210
Variable = .17.4.3.1.3.0.96.8.222.193.210
Value    = INTEGER - 3
```

- Step 5** Determine if you can view the link status through the ifOperStatus (2.2.1.8) variable. This is a two-step process:

- Look up the ifIndex used to index ifOperStatus in the dot1dBasePortIfIndex (17.1.4.1.2.2) entry.
- Look up the right ifOperStatus with the ifIndex.

The ifOperStatus return value should be 1 (Up):

```
C:\>snmputil get 192.168.3.221 public 17.1.4.1.2.2
Variable = .17.1.4.1.2.2
Value    = INTEGER - 61456
```

```
C:\>snmputil get 192.168.3.221 public 2.2.1.8.61456
Variable = interfaces.ifTable.ifEntry.ifOperStatus.61456
Value    = INTEGER - 1
```

- Step 6** Physically disconnect the client from the network element by pulling the cable out of the jack on the network element. The ifOperStatus return value should change to 2 (Down). If it does not, the network element can still be supported as a "generic - no link status" network element. The BayStack 303 falls into this category. The network element in this example, however, does support link status and is a generic network element.

```
C:\>snmputil get 192.168.3.221 public 2.2.1.8.61456
Variable = interfaces.ifTable.ifEntry.ifOperStatus.61456
Value    = INTEGER - 2
```

- Step 7** Check the dot1dTpAgingTime (17.4.2.0) implementation. In this example, the aging time has been set to 1800 seconds (30 minutes). A more typical value is 300 seconds (5 minutes).

```
C:\>snmputil get 192.168.3.221 public 17.4.2.0
Variable = .17.4.2.0
Value    = INTEGER - 1800
```

- Step 8** Change the network element's Aging Period through the Network Elements > Site *x* > Switches web page in WEBconfig. (Refer to the *Cisco BBSM 5.2 User Guide*.) You must use the read/write community string as the "password" on this web page, which is probably not public, although it is in this example. Repeat the "get" command used in [Step 3](#) and verify that the new dot1dTpAgingTime value is the one that you set. Although the network element can still be supported as a generic network element if the aging period is not programmable, the aging period configuration will not work.

- Step 9** Set the Aging Period to a low value. Ping from the client to reset the aging timer, disconnect the client, and monitor the dot1dTpFdbPort (17.4.3.1.2) entry for the client. It should disappear after the aging period expires.

```
C:\>snmputil get 192.168.3.221 public 17.4.3.1.2.0.96.8.222.193.210
Variable = .17.4.3.1.2.0.96.8.222.193.210
Value    = INTEGER - 2
```

When the aging period expires, you should see the following result when you check the dot1dTpFdbPort(17.4.3.1.2) entry.

```
C:\>snmputil get 192.168.3.221 public 17.4.3.1.2.0.96.8.222.193.210
Error: errorStatus=2, errorIndex=1
```

Some equipment (COM21) implements the dot1dTpAgingTime variable but expires the entries at the wrong time, which causes the connect time to be calculated incorrectly for network elements without link status support. Correct support of dot1dTpAging time is not important for network elements with link status, because the time that end user disconnects is without reference to the aging period for these network elements.

**Step 10** Configure the network element in WEBconfig and configure the port settings. Verify that you can connect through the network element and use the Active Ports report on the BBSM server to verify that the port number detected is correct. Physically disconnect the client from the network element. For generic network elements, the client should disappear from the Active Ports report in about 1 minute. For generic - no link status network elements, the aging period determines how long the client's entry stays in the Active Ports report.

**Step 11** Verify that a unique sysObjectID (1.2.0) value has been implemented. Along with changes to the BBSM registry, this unique sysObjectID is required for automatic network element detection.

```
C:\>snmputil get 192.168.3.221 public 1.2.0
Variable = system.sysObjectID.0
Value = OBJECT IDENTIFIER -
.iso.org.dod.internet.private.enterprises.43.10.27.4.1.2.1
```

Refer to the subkeys under the following directory for examples of registry entries that support auto detection of network element types and add them to the WEBconfig drop-down list:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IPOINT\Switches
```

The BayStack350 subtree shows how to support a generic network element, and BayStack303 shows entries for a network element without link status.

## Creating Your Network Element Module

Each BBSM network element module is contained in its own Microsoft MFC extension DLL. To generate the necessary project and source files to develop your network element DLL, the BBSM SDK provides custom application wizards.

Depending on your network element's requirements, your DLL will inherit from either the CEtherStack or CModemStack class:

- If the DLL inherits from the CEtherStack class, use the Cisco BBSM Ethernet Network Element Module AppWizard to create your MFC extension DLL project.
- If the DLL inherits from the CModemStack class, use the Cisco BBSM Modem Network Element Module AppWizard to create your MFC extension DLL project.

Both the Cisco BBSM Ethernet Network Element Module AppWizard and the Cisco BBSM Modem Network Element Module AppWizard generate a registry script file that is used to register your DLL when you build the pseudo-debug configuration.

For these procedures, you will need a BBSM 5.2 server with the Microsoft development environment and the BBSM SDK installed. For installation details, refer to [Chapter 2, "Getting Started."](#)

The following sections describe how to create your own network element module:

- [Using the Sample Network Element Policy Code, page 7-7](#)—Describes how to view a sample DLL and build a sample AccessPolicyBlock or AccessPolicyBlockASP project.

- [Developing and Debugging Your Network Element Module, page 7-8](#), describes how to develop and debug your module.
- [Integrating and Testing Your Network Element Module, page 7-12](#), describes how to build a release version of your DLL on your BBSM server, register your DLL on the BBSM server, and test the release version of your DLL with BBSM.

For helpful tips on debugging your module, refer to [Chapter 8, “Debugging Tips.”](#)

## Using the Sample Network Element Policy Code

This section describes how to view the sample implementation of the TUTMDULiteSwitch project provided in the SDK and how to build a sample DLL.

- 
- Step 1** Open the sample module workspace:
- Open **Microsoft Visual C++**.
  - From the menu, choose **File > Open Workspace**.
  - Browse to **c:\BBSM\_SDK** and select **SDK.dsw**.
  - Click **Open**. You now have the option of just viewing a sample project or using the code to build a sample DLL.
- Step 2** Within the workspace, navigate to the TUTMDULiteSwitch project and view the sample code in the Source & Headers folders.
- Step 3** If desired, build a sample TUTMDULiteSwitch DLL To build the DLL, you must stop BBSM services, build the project, then restart BBSM services:
- Stop BBSM services. Note that if you do not stop these services, you may not be able to build your project:
    - Open the Services window from **Start > Programs > Administrative Tools > Services**.
    - Stop each of the following services—AtDial, athdmn, WebPrint, TFTP, and PMSTestService. Right-click the service. Then, from the pop-up menu, click **Stop** and wait for the service to stop.
    - Stop these two services—IIS Admin Service and Microsoft ISA Server Control. Right-click the service. Then, from the pop-up menu, click **Stop**, then click **Yes**, and wait for the service to stop.
    - Close the Services window.
  - Build the TUTMDULiteSwitch DLL:
    - From the menu, choose **Build > Set Active Configuration**.
    - In the Set Active Project Configuration pane, click **TUTMDULiteSwitch – Win32 Pseudo-Debug** and then click **OK**.
    - Choose **Build > TUTMDULiteSwitch.dll**.
  - Restart BBSM services:
    - Choose **Start > Programs > Administrative Tools > Services**.
    - Restart each of the following services —AtDial, athdmn, WebPrint, TFTP, and PMSTestService. Right-click the service. Then, from the pop-up menu, click **Start** and wait for the service to start.

- Start each of these services—World Wide Web Publishing Service, FTP Publishing Service, Microsoft Web Proxy, and Routing and Remote Access. Right-click the service. Then, from the pop-up menu, click **Start** and wait for the service to start.
- Close the Services window.

## Developing and Debugging Your Network Element Module

Use the following procedure to develop and debug your module on your BBSM server.

- Step 1** Follow these steps to create your MFC extension DLL project.
- a. Open **Microsoft Visual C++**.
  - b. Choose **File > Open Workspace**.
  - c. Navigate to **c:\BBSM\_SDK** and double-click **SDK.dsw**.
  - d. Choose **File > New**.
  - e. In the New window, click the **Projects** tab.
  - f. Click one of the following:
    - **Cisco BBSM Ethernet Network Element AppWizard**
    - **Cisco BBSM Modem Network Element AppWizard**
  - g. In the Location field, navigate to **c:\BBSM\_SDK**.
  - h. In the Project name field, type the name of your module, such as **NewSwitch**.



**Note** This procedure assumes that your new MFC extension DLL project is named NewSwitch. Replace “New” with your network element’s name and append Switch. We use Switch as a naming convention, even though not all network elements are switches.

- Step 2** Click the **Add to Current Workspace** radio button, and then click **OK**.
- Step 3** Verify that the information presented in the New Project Information dialog box is correct, and then click **OK**.



**Note** The AppWizard generates the registry file: NewSwitch.reg. When you build the pseudo-debug configuration of your MFC extension DLL, this file registers your network element module.

- Step 4** Modify the NewSwitch.reg file with values specific to your module. Your new MFC extension DLL project contains a ReadMe.txt file that has details on modifying the registry values. For descriptions and information on modifying values, refer to [“Registry Entries for Network Element Modules” section on page 7-13](#).

- Step 5** Develop your MFC extension DLL.

Before you begin coding, determine how you will interrogate your network element to determine which port a client connects to and whether it is still connected. The answers to these questions determine what methods you must override in the base class provided in the SDK, CEtherStack. Usually you would use information in a private enterprise-specific MIB to obtain this information.

Each DLL implements a C++ class that supports one or more network element types. The registry contains an entry for each network element type. Network element types that have the same SNMP MIB structure, but differ in the number of ports or the sysObjectID value, can be supported by the same class but require separate registry entries.

The following substeps cover the methods that you are most likely to write or override. Refer to the [“Network Element Module API Reference” section on page 7-16](#) for detailed descriptions of these and other methods you may need to override.

a. Create a network element object instance.

All network element modules must provide a static method to create an instance of the C++ object that interfaces to the network element. The AppWizard generates this method for you. The object returned must be a subclass of `CEtherStack`, the base class for all network element modules. This method typically calls `new` to create an instance of the object and returns a pointer to it.

Your network elements must provide methods to do the following:

- Determine the location of client computers, given the client’s MAC and IP addresses
- Determine whether the client is still physically connected to the network element.
- Populate the port map table in the database.

b. Determine the client location.

The base class implements methods that use `dot1dTpDfDbPort` and `dot1dTpDfDbStatus` to determine the port number, given the client’s MAC address. If your network element module does not support these objects, your DLL must override the `FindPortFromMac` method.

The port identifier is a triple consisting of {stack number, switch number, port number}. If your network element does not support stacking (management of more than one device through a shared IP address), the switch number is always “1.” BBSM formats the port identifier as a string with 4 digits each for the stack and switch and 5 digits for the port number; for example, port 3 of switch 2 on stack 1 {1, 2, 3} has a port identifier “0001000200003.”

For a Cisco cluster, the Commander always has the `SwitchNumber = 1`. The rest of the cluster members have the `SwitchNumber = 2,3,...16`; for example, 0001000100003 = Commander, 0001000200003 = Member 1, 0001000300003 = Member 2, and so on.

We suggest using the `CEtherStack::FormatPortID` and `CEtherStack::ParsePortID` methods to format and parse port identifiers. You may need to provide your own formatting routines if you encode additional information, such as slot numbers, in the 5-digit port number.

c. Determine the client connection status.

The base class implements a method that uses `dot1dBasePortIfIndex` to convert the port number to an interface index and looks up the connection status in `ifOperStatus`. The base class expects the SNMP agent of your network element to return `Up` as the value of `ifOperStatus` when a powered client is connected to the port and `Down` when the client is powered down or disconnected. If your network element module does not support these objects, your DLL must override the `IsPortConnected` method.

If your network element determines the end of the session by waiting for the client’s SNMP forwarding database entry to expire, BBSM must subtract the aging time from the period between connection and detection to calculate the session duration accurately. Derive your class from `CModemStack` instead of `CEtherStack` for such network elements since the former performs this calculation automatically. (`CModemStack` is a subclass of `CEtherStack`.)

d. Populate the Port\_Map table.

The base class walks the dotIdTpPort table entry to discover the ports managed by the transparent bridge. It uses the number of ports configured in the registry as PortCount to mark port map entries for the network element as client ports. Ports marked as uplink ports are ignored during the search for client MAC address. The port map specifies the per-port policy for BBSM client sessions. Once the BBSM administrator configures the port with the same settings for all ports, they can use WEBconfig to customize the settings for an individual port.

Some network elements, such as DSLAMs, dynamically discover modems connected to it instead of providing a fixed number of ports. Your DLL implements the discovery algorithm to add entries for the modems to the port map by overriding the AddPortmapEntry method.

For Cisco 2950 and 3550 switches that have IOS 12.1(11) or later, use the RFC-1213 MIB variable ifDescriptor (.1.3.6.1.2.1.2.2.1.2) to count the FastEthernet and GigabitEthernet ports and populate the Port\_Map table in the database, because the dotIdTpPort table entries are present only for the ports that have a link status of Up.

e. Access the Port State table.

The PortRecordSet class, derived from CRecordset, encapsulates the Port\_State table in the BBSM server's SQL database. The m\_Client\_MAC\_address member contains the MAC address of the client. You may use this value to construct an object identifier (OID) to index a table in the private MIB of your network element. Refer to the [“Network Element Module API Reference” section on page 7-16](#) for details.

f. Access the Port\_Map table.

The PortMapRecordSet class, also derived from CRecordset, encapsulates the Port\_Map table in the BBSM server's SQL database. You may use the m\_Modem\_MAC\_addr member to store a value during port discovery in the AddPortMap method that will allow you to identify a cable or DSL modem during subsequent calls to FindPortFromMAC or FindPortFromIP.

For example, the code below is from a network element module's AddPortMap that discovers DSL modems. Notice that it obtains the virtual path and circuit identifiers for the ATM PVC and records this value in Modem\_MAC\_addr (the last argument to AddPortMapEntry).

```
// Walk the PVC table
// Look at all the active entries
PortMapRecordSet rPort(&db);
if (m_Agent.OpenSession())
{
    CString csBaseOid(m_apvcRowStatus);
    if (m_Agent.StartWalk(csBaseOid))
    {
        bOk = true;
        CString csOid;
        long lRowStatus;
        while (m_Agent.GetNextInteger(csOid, lRowStatus))
        {
            if (lRowStatus != ACTIVE)
                continue;
            // The PVC table is indexed by Vpi,Vci so the last two
            // octets of the OID are these values.
            // Look them up in the Modem_MAC_addr column.
            int nIndex = csOid.ReverseFind('.');
            if (nIndex >= 0)
            {
                nIndex = csOid.Left(nIndex).ReverseFind('.');
            }
        }
    }
}
```

```

if (nIndex <= 0)
continue;
{
CString csVpiVci = csOid.Mid(nIndex+1);
rPort.m_strFilter.Format("[Modem_MAC_addr] = '%s' \
    And [Site_number] = %d And [Port_ID] Like '%04d%%'",
csVpiVci, m_nSiteNumber, m_nStackNumber);
rPort.Requery();
if (!rPort.IsEOF())
continue;
// add a new portmap record
    CString csPortId = FormatPortID(m_nStackNumber, 1, nPort);
    nPort++;
AddPortmapEntry
    rPort
    0, // not an uplink
    *pnMinRoomNum,
    pPortMapArgs,
    csPortId,
    csVpiVci);

nPort++;
(*pnMinRoomNum)++'
nCount++;
    }
}
}
m_Agent.CloseSession();
}

```

Subsequently, a call to map the client computer MAC address to a BBSM port queries a private MIB table to get the PVC identifier, looks it up in the Port\_Map table, and thereby determines the location of the client.

```

bool bFound(false);
if (m_Agent.OpenSession())
{
long lVpi, lVci;
// Use MAC address to index bridgeAtmFdbVpi/Vci to get Vpi,Vci
    from the custom MIB
CString csOid(_T(m_bridgeAtmFdbVpi)+MACToOid(MACaddr));
CString csDebug;
// get the virtual path identifier
if (m_Agent.GetInteger(csOid, (long &lVpi))
{
// get the virtual circuit identifier
csOid = m_bridgeAtmFdbVci+MACToOid(MACaddr);
// get the virtual path identifier
if (m_Agent.GetInteger(csOid, (long &lVci))
{
// Now look up the Vpi,Vci pair in the modem mac
// address column
CString csVpiVci;
csVpiVci.Format("%01d.%01d", lVpi, lVci);
// Now look up the profile in the port map
CDatabase db;
if(DBASE_OPEN(db))
{

```

```

PortMapRecordSet rPort (&db);
rPort.m_strFilter.Format (" [Modem_MAC_addr] = '%s'
                        And \
                        [Site_number] = %d And [Port_ID] Like '%04d%'",
csVpiVci, m_nSiteNumber, m_nStackNumber);
try
    {
    if (rPort.Open())
        {
            if (!rPort.IsEOF())
                {
                    csPortId = rPort.m_Port_ID;
                    bFound = true;
                }
            rPort.Close();
        }
    }
catch(CDBException* e)
    {
    // error processing
    e->Delete();
    }

if (rPort.IsOpen())
    {
        rPort.Close();
    }
}
m_Agent.CloseSession();
}

```

- Step 6** Build the pseudo-debug configuration of your MFC extension DLL:
- a. From the Microsoft Visual C++ menu, choose **Build > Set Active Configuration**.
  - b. In the Set Active Project Configuration pane, click **NewSwitch – Win32 Pseudo-Debug**, and then click **OK**.
  - c. Choose **Build > Build NewSwitch.dll**. Your DLL is now built. It has been copied to c:\atcom\install\switches and registered using the generated registry file.
- 

## Integrating and Testing Your Network Element Module

The following are the steps for integrating and testing your network element module on your BBSM server.

- Step 1** Follow these steps to build the release configuration of your MFC extension DLL.
- a. On your server, open your MFC extension DLL project in Microsoft Visual C++.
  - b. From the menu, choose **Build > Set Active Configuration**.

- c. In the Set Active Project Configuration pane, click **NewSwitch – Win32 Release**, and then click **OK**.
  - d. Choose **Build > Build NewSwitch.dll**.
- Step 2** On your BBSM server, copy the release network element files to the appropriate directories. (See [Table 7-2](#).)

**Table 7-2 Copying the Release Network Element Files**

Copy the release network element files from:	To the directories on your BBSM server:
c:\BBSM_SDK\NewSwitch\release\ NewSwitch.dll	c:\atcom\install\switches
c:\BBSM_SDK\NewSwitch\NewSwitch.reg	c:\atcom

- Step 3** Register your MFC extension DLL on your BBSM server:
- a. Open Windows Explorer.
  - b. Navigate to **c:\atcom** and double-click **NewSwitch.reg**.
  - c. To add information to the registry, click **Yes**. A dialog box appears, stating that the information has been successfully entered. Click **OK**.

- Step 4** Verify that your network element module is installed and working on your BBSM server.

A test plan for a completed network element module should contain at least the items listed below. Turn on the Trace feature and review the Debugging table contents to verify that your module is behaving correctly. Refer to [Chapter 8, “Debugging Tips.”](#) For descriptions of the AtDial tables, refer to [AtDial Database Schema, page A-1](#).

- WEBconfig displays your new network element in the Switch Type list on the Switches web page.
- WEBconfig configures the port correctly for your network element through the Port Settings web page in WEBconfig and by using the Port Control feature. Examine the Port\_Map table to verify this.
- Connect a client through your network element and establish an active session. Run the Active Ports report to verify that BBSM has correctly identified the client port.
- Physically disconnect the client computer from the network component. Refresh the Active Ports report until the client computer disappears. Verify that this time is either within 1 minute of disconnection or within 1 minute plus the aging time, depending on the IsPortConnected implementation.
- Examine the Transaction\_History table to verify that the calculated session duration is correct.

## Registry Entries for Network Element Modules

The BBSM server obtains the location and descriptions of network element modules from the Windows 2000 registry tree rooted in the following location:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IPOINT\Switches
```

The following typical registry editor script would be used to insert network element information under this key:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IPORT\Switches\
  TutMDULite]
"Code"=dword:0000001a
"CreateStack"="CreateTutMDULite"
"Description"="Tut Systems Espresso MSU Lite"
"DLLPath"="c:\\atcom\\install\\switches\\TutMDULiteSwitch.dll"
"SysObjectID"="1748.2.4"
"PortCount"=dword:00000008
"SwitchModeList"="1Mbps"
"PortmapOnTheFly"=dword:00000000
"PortType"=dword:16
"SwitchCapability"="STAND_ALONE"
```

This entry configures the registry for a network element DLL that supports the Tut Systems Espresso MDU Lite DSL-like solution.

### Code

The Code is a unique arbitrary number used by BBSM to index the table of network elements. The AppWizard generates this unique value by scanning the existing registry entries. The value should be unique among all the network element modules in the HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\IPORT\Switches key and should be small positive number. The generated value is 1 plus the highest pre-existing value.




---

**Note** Use the generated value. If you choose to change this value, make sure the value is unique.

---

### CreateStack

CreateStack is the entry point used to create an instance of the C++ object used to query the network element. The AppWizard generates this value and places the matching name in the .reg file. If you change the name of this method, you must also change the value in the registry file.

### Description

The WEBconfig Switches page displays the description in the Switch Types drop-down menu. Modify this value with a description of the network element module.

### DLLPath

The DLLPath is the full path to the MFC Extension DLL that provides the logic that supports the particular network element type.

### PortCount

The PortCount is the number of ports on the device to which you could connect clients. Change this value to the number of client (non-uplink) ports on your network element. Some devices, such as DSLAMs, do not have a fixed number of ports. The number of “ports” on a DSLAM depend on how many DSL modems connect through it. Set “PortCount” to zero for such systems.

**PortmapOnTheFly**

The PortmapOnTheFly variable is used to dynamically generate the port map for a CMTS:

- When you are using a CMTS (cable modem), set this variable to 1.
- For the rest of switches and access points, the variable should be set to 0.

**PortType**

The PortType identifies the RADIUS NAS (RADIUS Client) port type. Set this variable according to the network element type you are using, as follows:

- 15 = Ethernet
- 16 = xDSI
- 17 = Cable
- 19 = 802.11

Set it to 5 = Virtual, if you use the Null switch type. (Refer to the switches registry for a Null stack entry.)

**SwitchCapability**

The SwitchCapability variable identifies the cluster capability of the network element:

- For Cisco cluster-capable switches, set the variable to CLUSTER\_CAPABLE.
- For 3Com and Cabletron switches, set it to STACK\_CAPABLE.
- For the non-Cisco switches, access points, and CMTSs, set it to STAND\_ALONE.

**SwitchModeList**

The SwitchModeList identifies the switch mode to be used during port testing, which is done through the WEBconfig interface. Modify this value with the rate appropriate for your network element configuration. This value determines the parameter values used for port testing. (See [Table 7-3](#).)

**Table 7-3** Switch Mode Settings

Switch Mode (Mbps)	Pings to Send (bytes)		Interpacket Delay (ms)		Echo Data Size (bytes)	
	Default	Range	Default	Range	Default	Range
1	500	300–700	85	85–90	1024	768–1280
5			45	45–50		
10			10	10–15		
15			8	6–9		
100			3	1–5		

The SwitchModeList value is used to populate the switch modes on the Port Tests page of WEBconfig. If you want more than one switch mode to be available from the Port Tests page, you must define this value in hex. For example, to have 10 Mbps and 100 Mbps available in the Switch Mode list on the Port Tests page, your regedit script would have the following line:

```
"SwitchModeList"=hex(7):31,30,4D,62,70,73,00,31,30,30,4D,62,70,73,00,00
```

### SysObjectID

The SysObjectID is the value of a specific entry in the hardware device SNMP MIB defined in RFC 1213, which identifies the type of hardware. BBSM uses this information to determine the type of switching hardware in use. This example shows a single sysObjectID, represented as a registry value of type string. If functionally equivalent network elements respond with different sysObjectIDs, you can use a multi-string value to encode the two or more sysObjectIDs. Modify this value with the value returned by an SNMP query of sysObjectID from your network element module. Omit the leading 1.3.6.1.4(private) prefix.

## Network Element Module API Reference

This section provides a reference that you can use to create your own network element module. Understanding these classes and their methods is critical to successfully creating your module. The section focuses on the methods exposed by the base class, [CEtherStack Class, page 7-16](#), which you override to implement your network element module. The base class provides support for network elements that comply with the industry standards for transparent bridges. You must override the behavior of the base class where your network element diverges from this standard.

The section also documents the CModemStack and CSNMPAgent classes that you might call from your network element class:

- [CModemStack Class, page 7-33](#)
- [CSNMPAgent Class, page 7-35](#)

## CEtherStack Class

The CEtherStack class implements support for transparent bridges conforming to the standards described in preceding sections. These standards specify SNMP MIBs that enumerate ports that learn MAC addresses (the transparent port table), map client MAC addresses to ports (the transparent bridge forwarding database) and report the link status on the port (the Interfaces table).

Not all network elements or even transparent bridges implement these standards. This section documents the methods that you must override to support network elements that do not implement or only implement part of these standards.

The methods are listed in alphabetical order. Some are methods that you may override, and others are methods you call from your derived calls. The following are the key methods to understand:

- The constructor
- AddPortMap
- CreateStack
- IsPortConnected
- GetPortFromMac

## Class Enumerations

This enumeration documents the values of MIB objects used by the `CEtherStack` class. Refer to the specifications (RFCs) for details. The following are the values for `ifOperStatus`. Refer to RFC 1213.

```
enum
{
    UP = 1,
    DOWN = 2,
    TESTING = 3
};
```

The following are the values for `dot1dTpFdbStatus`. The class ignores forwarding database entries with status values of `INVALID` and `SELF`.

```
enum RFC1493_PORT_STATUS
{
    RFC1493_PORT_STATUS_OTHER = 1,
    RFC1493_PORT_STATUS_INVALID,
    RFC1493_PORT_STATUS_LEARNED,
    RFC1493_PORT_STATUS_SELF,
    RFC1493_PORT_STATUS_MGMT
};
```

## Member Variables

The `CEtherStack` class includes the following member variables.

`CSNMPAgent` **`m_Agent`**

This object implements the interface to the SNMP agent associated with the network element. The IP address and SNMP community string for this agent are passed to the constructor of the `CEtherStack` class. This class is a wrapper around the Microsoft SNMP API. See the Microsoft Platform SDK documentation for details on this API.

```
static const char* m_dot1dBasePortIfIndex
static const char* m_dot1dTpFdbPort
static const char* m_dot1dTpFdbStatus
static const char* m_dot1dTpPort
```

These four variables contain the SNMP object identifiers for these managed objects. `CEtherStack` passes these strings to methods of the SNMP agent object to query that agent's MIB. If the string begins with a period, it is the full object identifier for this object. If it does not begin with a period, the prefix “.1.3.6.1.2.1” (MIB-II) is implied.

`int` **`m_nSiteNumber`**

The site number associated with the network element. The BBSM internal network can be logically divided into multiple sites; each site is assigned a site number.

`int` **`m_nStackNumber`**

The stack number associated with the network element. Stack numbers must be unique within a site. The stack number is equivalent to the cluster number. A cluster can have up to 16 member switches or network elements. There can be a cluster with only one network element.

`int` **`m_nSwitchNumber`**

The switch number associated with the network element. Switch numbers are unique within a cluster. The cluster Commander switch always has the switch number 1. The rest of the cluster members have switch numbers from 2 to 16.

## CEtherStack::CEtherStack Method

The CEtherStack method calls the constructor for CEtherStack from your sub-class constructor. Your constructor is usually called from static “CreateStack” method implemented by your class which the BBSM core calls to create an instance of your network element module interface object.

```
CEtherStack
(
    const CString& csIP,
    const CString& csPassword,
    int nSiteNumber,
    int nStackNumber = 1
    int nSwitchNumber = 1
);
```

### Parameters

*csIP*

IP address of the SNMP agent on the network element.

*csPassword*

SNMP community string (read or read/write) from the network element.

*nSiteNumber*

This number associates the network element with a site in multi-site applications.

*nStackNumber*

Each network element within a site is assigned a stack number. This number is the first 4 digits of the port identifier within a site. As of BBSM software release 5.2, the stack number is equivalent to the cluster number.

*nSwitchNumber*

Each network element within a cluster is assigned a switch number. The number is the second set of 4 digits in the port identifier within a site.

## CEtherStack::~~CEtherStack() Method

The virtual destructor insures that your derived class destructor is called when an object is deleted through a base class pointer.

```
virtual ~CEtherStack();
```

There are no parameters for this method.

## CEtherStack::AddDefaultPortmap Method

In the base class, the AddPortMap implementation queries the network element to determine the port count. In contrast, the AddDefaultPortmap() method supports port map generation when a network element is **not** present. This method returns true if default port map generation succeeds.

The base class implementation generates port map entries, as described in the AddPortMap section, for the number of ports specified in the registry value PortCount. The nSwitchCount parameter is obtained by the core from the GetSwitchCount method.

Override this method to implement a different port map generation algorithm when the network element is not connected.



#### Note

Not all network elements have sensible default values for the port map table. DSL and cable modem networks, for example, generate a port for each modem and BBSM cannot determine what modems are connected to such systems if the BBSM is not connected to them. This method should always return false in such cases. Port map generation in the absence of the network element hardware is discouraged.

```
virtual bool AddDefaultPortmap
(
    int          nSwitchCount,
    PINT        pnMinRoomNum,
    const PortMapArgs& pPortMapArgs,
);
```

#### Parameters

*nSwitchCount*

Number of switches in stack.

*pnMinRoomNum*

Pointer to room number, incremented for each map entry created.

*struct pPortMapArgs*

Pointer to a structure of the following parameters:

*CTime ctStartAuthTime*

Start of authorize period for subscription access model

*Ctime ctEndAuthTime*

End of authorize period for subscription access model

*LPCSTR pszPageSet*

Page set to configure for each port map entry created

*LPCSTR pszStartPage*

Initial page to display to end user connecting to this port

*UINT nBandwidthKbps*

The default bandwidth for each port map entry created

*UNIT nMultinetNumber*

The multinet number: 1 if configured for singlenet or 2 if configured for multinet

*long nToolbarNumber*

Deprecated parameter

*long nToolbarMode*

Deprecated parameter

*long nToolbarFrequency*

Deprecated parameter

*long nToolbarTime*

Deprecated parameter

*BOOL bEnablePortHop*

Enables or disables the Port Hop feature that allows an end user to move between ports without reauthenticating

*long lSiteNumber*

The current site number

*LPCSTR pszRoomNumber*

The first room number will be incremented by one during AddPortmap process.

*LPCSTR pszSwitchMode*

Select the desired rate in Mbps within the ranges that the switch is able to support: 1Mbps, 5Mbps, 10Mbps, 15Mbps, or 100Mbps. The default is 10Mbps.

*long lPingsToSend*

The range is 300 to 700, and the default is 500.

*long lEchDataSize*

The desired size in bytes for the echo data. The range is 768 to 1280, and the default is 1024.

*long lInterPacketDelay*

The desired interpacket delay in milliseconds (ms). (See [Table 7-4](#).)

**Table 7-4 Inter Packet Delay Switch Mode Options**

Switch Mode (Mbps)	Inter Packet Delay (ms)	
	Default	Range
1	85	85–90
5	45	45–50
10	10	10–15
15	7	6–9
100	3	1–5

*CTime C\_Time\_of\_last\_configure*

The time the port was last configured

## CEtherStack::AddPortmap Method

When the BBSM administrator uses the WEBconfig PortMap page to generate a port map for a site, WEBconfig calls this method for all network elements associated with that site. At the end of this procedure, new port map entries created by WEBconfig will have sequential “dummy” room number and identical access and accounting policies. Subsequent steps in the BBSM configuration (outside the scope of this document) replace the “dummy” room numbers with actual room numbers and potentially configure different policies on different ports.

In programming terms, WEBconfig calls AddPortMap for each row of the Switches table where SiteNumber is equal to the selected site number. AddPortMap should enumerate the ports on the network element. For all ports not presently in the port map, it should call AddPortMapentry to create a new port map entry, passing the arguments supplied to it to set the initial value for the port map entry. See the [“CEtherStack::AddPortmapEntry Method” section on page 7-21](#).

The pnMinRoomNum parameter points to an integer that this method should increment for each map entry created. WEBconfig passes the updated value to the next network element interface module to ensure that each module generates unique dummy room numbers.

The base class implementation walks the dot1dTpPort entry of the transparent port table in the bridge MIB. This enumerates all the ports that learn MAC addresses on a transparent bridge.

Override this method if your network element uses a different method to discover the ports to which a client might connect.

```
virtual bool AddPortmap
(
    PINT          pnMinRoomNum,
    const PortMapArgs& pPortMapArgs,
);
```

## Parameters

*pnMinRoomNum*

Pointer to room number, incremented for each port map entry created.

*pPortMapArgs*

Pointer to a structure of the same parameters described in the [“CEtherStack::AddDefaultPortmap Method” section on page 7-18](#).

## CEtherStack::AddPortmapEntry Method

Use this method in the AddPortMap method of your derived class to add rows (port map entries) to the Port\_Map table in the AtDial database. The goal of AddPortMap is to initialize the Port\_Map table with unique location identifiers using the same access and account policy for all ports. Subsequent configuration operations may set special policies for some ports and supply real room numbers.

Use FormatPortID to create the csPortID string from the stack number (m\_nStackNumber), the switch number (m\_nSwitchNumber) and the port number. Also see the MIBtoMap and MaptoMIB methods.

AddPortMap calls AddPortMapentry each time it adds an entry to the port map. The base class implementation of AddPortMap does not use the pszModemMACaddr parameter, so the value of the pszModemMACaddr defaults to NULL. We provide this parameter to support network module implementations that need to save information in the port map to identify the port connected to a client in a non-standard way. AddPortMapEntry writes the string pointed to by pszModemMACaddr to the ModemMACaddr of the Port\_Map table. Refer to the port map table in the [“AtDial Database Schema” section on page A-1](#).

An example of how you might use `pszModemMACaddr` is as follows: assume that you are implementing a network module that supports a cable modem plant and that you can obtain the MAC address of the cable modem to which a client attaches. If you store the MAC address of the cable modem in this field, you can look up this address in the `Port_Map` table to find the port map entry associated with this cable modem.

```
bool AddPortmapEntry
(
    PortMapRecordSet& rPort,
    BOOL              nUplinkPort,
    int               nRoomNum,
    const PortMapArgs& pPortMapArgs,
    const CString&    csPortID,
    const char*       pszModemMACaddr=NULL
);
```

## Parameters

*rPort*

A derived class of `CRecordset`, which is used to add rows to the port map table.

*nUplinkPort*

0 (false) if connected to end user devices, 1 (true) if connected to other network elements.

*nMinRoomNum*

Room number or location identifier, incremented by caller for each port map entry created.

*pPortMapArgs*

Pointer to a structure of the same parameters described in the [“CEtherStack::AddDefaultPortmap Method” section on page 7-18](#).

*csPortID*

This parameter supplies the identifier of a port to which the client may connect. The format of this string is `XXXXYYYYZZZZZ`, where `XXXX` is the 4-digit stack number, `YYYY` is the 4-digit switch number, and `ZZZZZ` is the 5-digit port number. The network elements that do not support stacking always set `YYYY` to `0001`.

*pszModemMACaddr*

String used to identify a port.

## CEtherStack::DeletePortmap Method

The `DeletePortmap` method removes all port map entries associated with this network element from the port map. `WEBconfig` calls this method when the BBSM administrator changes the port settings for a network element before calling `AddPortMap`. Because the logic in this method does not depend on the network element implementation, there is no reason to override it.

```
virtual void DeletePortmap();
```

There are no parameters for this method.

## CEtherStack::FindPortFromIP Method

The FindPortFromIP method calls the CEtherStack::GetMacFromIP to obtain the client MAC address from the IP address and then calls CEtherStack::FindPortFromMac. It is not called from the BBSM core but is available for use by third-party utilities. It returns true if a client with CSIP is found on the network element. You should probably not override this method. Instead, implement a custom FindPortFromMAC method to support network elements that do not support the transparent bridge MIB.

```
virtual bool FindPortFromIP(CString& csPortId, const CString& csIP);
```

### Parameters

*csPortId*

When this method returns true, it returns the identifier of the port to which the client is connected. The format of this string is XXXXYYYYZZZZZ, where XXXX is the 4-digit stack number, YYYY is the 4-digit switch number, and ZZZZZ is the 5-digit port number. The network elements that do not support stacking always set YYYY to 0001.

*csIP*

This parameter supplies the IP address of the client.

## CEtherStack::FindPortFromMac Method

If a client is connected to a network element, this method uses the MAC address of the client to determine the port to which the client is connected and returns true. If the client is not connected to the network element, this method returns false.

The BBSM core software calls this method to get the port identifier for an end user session. It uses this information to query the port map to find the start page to display to clients connected to that port.

The base class uses the MAC address as an index into the transparent bridge table entry dot1dTpFdbPort. If this entry exists, the port number in the MIB is used to calculate the port number in the map. See the MIBtoMap method. The port number in the map is concatenated with the stack number (*m\_nStackNumber*) and switch number (*m\_nSwitchNumber*) to produce the port identifier.

Override this method if your network element MIB does not support the dot1dTpFdbPort managed object.

```
virtual bool FindPortFromMac(CString& csPortId, const MACADDR& MACaddr);
```

### Parameters

*csPortId*

When this method returns true, *csPortId* returns the identifier of the port to which the end user is connected. The format of this string is XXXXYYYYZZZZZ, where XXXX is the 4-digit stack number, YYYY is the 4-digit switch number, and ZZZZZ is the 5-digit port number. The network elements that do not support stacking always set YYYY to 0001.

*MACaddr*

This parameter supplies the client's 6-byte MAC address.

## CEtherStack::FormatPortID Method

The FormatPortID method accepts the components of a port identifier and returns a string in port identifier format. The format of this string is XXXXYYYYZZZZZ, where XXXX is the 4-digit stack number, YYYY is the 4-digit switch number, and ZZZZZ is the 5-digit port number. The network elements that do not support stacking always set YYYY to 0001. Call this method instead of formatting the port identifier yourself since the format could change in a future release.

See the “[CEtherStack::ParsePortID Method](#)” section on page 7-31.

```
static CString FormatPortID(int nStack, int nSwitch, int nPort);
```

### Parameters

*nStack*

This parameter is the stack number of the port (XXXX).

*nSwitch*

This parameter is the switch number within the stack (YYYY).

*nPort*

This parameter is the port number within the switch (ZZZZZ).

## CEtherStack::GetAdminStatus Method

The GetAdminStatus method reports the administrative status of the network element port. If nAdminStatus is UP, packets are allowed to pass through the port. If the value is DOWN, transmission of packets through this port is prohibited. Behavior for other values of nAdminStatus is undefined. The network element port will not pass packets and the client cannot access the network if the port is disabled.

The base class gets the value of the ifAdminStatus object in the network element MIB, as described in RFC-1213. Override this method if your network element does not support this interface.



### Note

This method supported a discontinued product, and, currently, the BBSM core does not call it. We do recommend, however, that you implement reporting of administrative status to support features that Cisco or third parties may implement in a future release.

```
virtual bool GetAdminStatus(long nIfIndex, long& nAdminStatus);
```

### Parameters

*nIfIndex*

The interface index of the port (see RFC-1213).

*nAdminStatus*

Returns the Administrative status of the port associated with the interface index, n. If Index

## CEtherStack::GetAgingTime Method

The base class implementation reads the value of dot1dTpFdbPort from the MIB of the network element. If this method successfully retrieves the value, it returns true. The caller of this method **must** check the return value since the value of lSeconds is undefined if the return value is false.

See the “[CEtherStack::FindPortFromMac Method](#)” section on page 7-23 and RFC-1493 for a discussion of the transparent bridge forwarding table.

Override this method if the network element reports the aging time of MAC to port mappings in a different way. If your network element does not report the aging time, always return false.

```
virtual bool GetAgingTime(long& lSeconds);
```

### Parameters

*lSeconds*

The number of seconds that a MAC address entry remains in the forwarding table of the transparent bridge

## CEtherStack::GetClientPortCount Method

For the base class, this method returns the values entered on Switches page of WEBconfig. If *nSwitch* is “1,” the base class implementation of `GetClientPortCount` returns the value entered on the Switches page in WEBconfig in the Client Ports field for switch 1. If *nSwitch* is not “1,” it returns the value entered on switch 2-*n*.

`CEtherStack::AddPortMap` sets the uplink flag to false (zero) for the first `GetClientPortCount(1)` ports and to true for the remainder of the ports.

You may override this class to return a different value, but generally the system should use the client port count configured by the Administrator through the BBSM Dashboard.

```
virtual int GetClientPortCount(int nSwitch);
```

### Parameters

*nSwitch*

The switch number within the stack

## CEtherStack::GetDisconnectTimeAdjustment Method

The BBSM core calculates the duration of a session by subtracting the time the end user authenticated from the time that the end user terminates the session. If the end user terminates the session by disconnecting from the network element, some network elements do not detect this immediately. For example, a network element that does not support `ifOperStatus` as described in the `IsPortConnected` section might not know that the end user disconnected until the MAC entry in the forwarding database ages out.

Thus `IsPortConnected` may return false some time after the end user disconnects and the session duration must be adjusted for this delay. The BBSM core calculates the session duration as:

```
time(IsPortConnected() == false) - time(end user authenticates) -  
GetDisconnectTimeAdjustment()
```

Because the value `ifOperStatus` updates immediately, the base class implementation of this method always returns zero. See the “[CModemStack Class](#)” section on page 7-33 for a description of an alternate implementation.

Override this class if your derived class needs to adjust the session duration.

```
virtual int GetDisconnectTimeAdjustment();
```

There are no parameters for this method.

## CEtherStack::GetMacFromIP Method

The BBSM core calls this routine to map an end user IP address to its associated MAC address. The base class implementation looks up the router associated with the IP address in the Port\_State table. Next, it looks for a mapping from the IP address to the MAC address in the ipNetToMediaPhysAddress entry of the ipNetToMedia table. See RFC-1213. The BBSM core uses the MAC address in the call to GetPortFromMac.

This method returns true if a client with MACaddr found on the network element. Override this class to support network element modules that provide a different way of obtaining the port number from the client MAC address.

```
static bool GetMacFromIP
(
    int& nRouterNumber,
    MACADDR& MACaddr,
    const CString& csIP
);
```

### Parameters

*nRouterNumber*

This parameter returns the value of RouterNumber for the row in the Routers table associated with the IPaddress (output).

*MACaddr*

This parameter returns the MAC address corresponding to the IP address (output).

*csIP*

The IP address to resolve as a MAC address.

## CEtherStack::GetOperStatus Method

The GetOperStatus method should return true if a client is connected to the network element port identified by nIfIndex. The base class queries the ifOperStatus object in the MIB and returns true if the value of this object is UP.



### Note

This method supported a discontinued product and currently, the BBSM core does not call it. We recommend, however, that you implement this method to support feature that Cisco or third parties may implement in a future release.

```
virtual bool GetOperStatus(long nIfIndex, long& nOperStatus);
```

### Parameters

*nIfIndex*

The interface index of the port (see RFC-1213).

*nOperstatus*

The value of ifOperStatus.

## CEtherStack::GetOperStatusFromPort Method

The base class calls this method from within `CEtherStack::IsPortConnected` to determine if the client is still connected to this port on the network element. You probably will not call this method directly. BBSM considers the device connected if `lStatus` is UP.

```
bool GetOperStatusFromPort(int nMIBPort, long &lStatus);
```

### Parameters

*nMIBNumber*

The port number as defined in the bridge MIB. See RFC-1493.

*lStatus*

The value of `ifOperStatus`.

## CEtherStack::GetSwitchesPerStack Method

The Switch Discovery utility, part of the BBSM core, calls this routine during port map generation to determine whether to probe for switches sharing the same management IP address. The base class always returns one since it does not support stacked switches. Override this method to support stackable switches.

```
virtual int GetSwitchesPerStack() const;
```

There are no parameters for this method.

## CEtherStack::GetSysObjectID Method

Call the `GetSysObjectID` method to get the `sysObjectID` value in the MIB associated with a network element. This value is defined by the network element vendor. This information can be useful in TRACE debugging statements. This is not a virtual method, so you cannot override it.

```
CString GetSysObjectID();
```

There are no parameters for this method.

## CEtherStack::GetTotalPortCount Method

This method is obsolete and may be removed from a subsequent release. The BBSM core does not call it.

```
virtual int GetTotalPortCount();
```

There are no parameters for this method.

## CEtherStack::GetUsageCounts Method

The `GetUsageCounts` method returns cumulative statistics on network traffic on the port identified by `nIfIndex`. If this information is not available from the network element, the method returns false. The base class queries the `ifInOctets`, `ifInErrors`, `ifOutOctets` and `ifOutErrors` objects in the network element MIB, as described in RFC-1213.

Override this method if your network element does not provide usage information through the MIB objects described above. If your network element does not support collection of usage information, your method should always return false.

**Note**

This method supported a discontinued product and currently, the BBSM core does not call it. We recommend, however, that you implement collection of these statistics to support usage based accounting policies that Cisco or third parties may implement for a future release.

```
virtual bool GetUsageCounts
(
    long nIfIndex,
    unsigned long& uInOctets,
    unsigned long& uInErrors,
    unsigned long& uOutOctets,
    unsigned long& OutErrors
);
```

**Parameters**

*nIfIndex*

The interface index of the port (see RFC-1213).

*uInOctets*

Returns the number of bytes received from the client(s).

*uInErrors*

Returns the number receive errors from the client(s).

*uOutOctets*

Returns the number of bytes sent to the client(s).

*uOutErrors*

Returns the number transmit errors to the client(s).

**CEtherStack::IsPortConnected Method**

The BBSM core polls this method during an active session to determine if the client is still connected to the network element. It returns true if the network element reports that the client with IP address *csClientIP* is still connected to the port identified by *csPortId*. The base class queries the *ifOperStatus* object in the network element MIB to determine whether the interface is UP. Refer to RFC-1213 for the MIB definition that specifies this object. The base class ignores the *csClientIP* parameter, but we provide it in the signature because we expect some subclasses to need this information to determine the client connection status.

A consequence of this design is that, for the base class, a client connected through a hub or other device will appear to the BBSM core to remain connected even if the client disconnects from the hub as long as the hub remains connected to the network element port.

This method assumes that the *ifOperStatus* will report UP if any client is connected to the port and DOWN if no client is connected to the port. This is not true of all implementations: some network elements report UP unless the port fails, regardless of whether a client is connected to the port. If this is the only divergence from the transparent bridge model, you can use the *CNoLinkStatus* class to support your network element. Create a registry entry as described the “RFC Compliant Components” section. Set values *DLLPath* to *c:\atcom\install\switches\GenericSwitchNoLinkStatus.dll* and *CreateStack* to *CreateNoLinkStatus*.

If your network element does not implement `ifOperStatus` or it does not report UP and DOWN as described above and you cannot use `CNoLinkStatus`, override this method.

```
virtual bool IsPortConnected(const CString& csPortId, const CString& csClientIP);
```

## Parameters

*csPortId*

This parameter supplies the identifier of the port to which the client is connected. The format of this string is `XXXXYYYYZZZZZ`, where `XXXX` is the 4-digit stack number, `YYYY` is the 4-digit switch number, and `ZZZZZ` is the 5-digit port number. The network elements that do not support stacking always set `YYYY` to `0001`.

*csClientIP*

This parameter supplies the IP address of the client.

## CEtherStack::IsSwitchConnected Method

The `IsSwitchConnected` method determines whether the network element is connected to the internal BBSM network. For stackable switches (multiple switches sharing an IP address), the `nSwitchNumber` designates the individual switch in the stack to poll.

The base class implementation does not support stacking so it returns false if the `nSwitchNumber` is not "1." If the `nSwitchNumber` is "1," it queries `sysServices` in the SNMP MIB to see if the agent on the network module responds. If the agent responds, it returns true; otherwise it returns false.

Override this method to support stackable switches or if your network element does not support the SNMP query performed by the base class.

```
virtual bool IsSwitchConnected(int nSwitchNumber);
```

## Parameters

*nSwitchNumber*

The switch index within the stack.

## CEtherStack::MACToOid Method

Some tables in the MIBs used by BBSM are indexed by a MAC address. To access an element in such a table, one appends the MAC address in OID format to the OID for the element. For example, to look up the port number that has learned a MAC address, one appends the MAC address to `dot1dTpFdbPort` as in the following:

```
CString csPortNumberOid(m_dot1dTpFdbPort + MACToOid(MACaddr));
static CString MACToOid(const MACADDR& MACaddr);
```

## Parameters

*MACaddr*

The MAC address to convert to an SNMP object identifier

## CEtherStack::MACtoUnpacked Method

The MACtoUnpacked utility accepts a MAC address in binary format and returns a string in formatted as six pairs of hexadecimal digits (i.e.: “00 10 5a 82 2c fa”), each pair representing a byte. The BBSM core uses this routine to convert MAC addresses in binary form to text to store in the database or place in TRACEx statements. See the “[CEtherStack::UnpackedToMAC Method](#)” section on page 7-33.

```
static CString MACtoUnpacked(MACADDR& MACaddr);
```

### Parameters

*MACaddr*

The MAC address in binary form (input)

## CEtherStack::MaptoMIB Method

As described in the CEtherStack::AddPortMap section, the base class implementation walks the transparent port table to identify the ports that learn MAC addresses on the transparent bridge. For most vendors, the port number in the MIB matches the physical port number on the network element. Some networking products return inconvenient numbers, which do not match the physical port.

The base class returns the value passed to it. If the value in the port map does not match the value in the MIB, BBSM must convert it to correctly access the MIB. Override this function to map port numbers in the port map table to numbers in the MIB.

The method should return false if nMapNumber does not map to a port number in the MIB.

See the “[CEtherStack::MIBtoMap Method](#)” section on page 7-30.

```
virtual bool MaptoMIB(int nMapNumber, int& nMIBNumber);
```

### Parameters

*nMapNumber*

The port number as defined in the port\_map table.

*nMIBNumber*

The port number as defined in the bridge MIB. See RFC-1493.

## CEtherStack::MIBtoMap Method

As described in the CEtherStack::AddPortMap section, the base class implementation walks the transparent port table to identify the ports that learn MAC addresses on the transparent bridge. For most vendors, the port number in the MIB matches the physical port number on the network element. Some networking products return inconvenient numbers, which do not match the physical port. The base class returns the value passed to it. BBSM operators generally prefer that the port number in the identifier found in the Port\_map match the physical port. Override this function to map port numbers in the MIB to physical port numbers. The method should return false if nMIBNumber does not map to a physical port. Refer to the “[CEtherStack::MaptoMIB Method](#)” section on page 7-30.

```
virtual bool MIBtoMap(int nMIBNumber, int& nMapNumber);
```

### Parameters

*nMIBNumber*

The port number as defined in the bridge MIB. See RFC-1493.

*nMapNumber*

The port number as defined in the port\_map table.

## CEtherStack::ParsePortID Method

The ParsePortID method parses the port identifier string and writes the value of the components through the pointer arguments. Use this method instead of parsing the port identifier yourself since the port identifier format could change in a future release. See the “[CEtherStack::FormatPortID Method](#)” section on page 7-24.

```
static void ParsePortID
(
    const CString& csPortID,
    PINT pnStack,
    PINT pnSwitch,
    PINT pnPort
);
```

### Parameters

*csPortID*

This parameter supplies the identifier of a port to which the end user may connect. The format of this string is XXXXYYYYZZZZZ, where XXXX is the 4-digit stack number, YYYY is the 4-digit switch number, and ZZZZZ is the 5-digit port number. The network elements that do not support stacking always set YYYY to 0001.

*pnStack*

This parameter points to the location to receive the stack number of the port (XXXX).

*pnSwitch*

This parameter points to the location to receive the switch number within the stack (YYYY).

*pnPort*

This parameter points to the location to receive the port number within the switch (ZZZZZ).

## CEtherStack::SetAdminStatus Method

The SetAdminStatus method enables the network element port if nAdminStatus is UP and disables if the value is DOWN. Behavior for other values of nAdminStatus is undefined. The network element port will not pass packets and the end user cannot access the network if the port is disabled.

The base class sets the value of the ifAdminStatus object in the network element MIB, as described in RFC-1213. Override this method if your network element does not support this interface. This method will fail and return false unless the csPassword parameter passed to the constructor is the SNMP read/write community string for the network element’s SNMP agent.



### Note

This method supported a discontinued product and, at this writing, the BBSM core does not call it. It is recommended that you support controlling the administrative status of port on your network element to support features that Cisco or third parties may implement in a future release.

```
virtual bool SetAdminStatus(long nIfIndex, long nAdminStatus);
```

## Parameters

*nIfIndex*

The interface index of the port (see RFC-1213)

## CEtherStack::SetAgingTime Method

The base class sets the value of dot1dTpDfDbPort in the MIB of the network element. If this method successfully retrieves the value, it returns true. The caller of this method must check the return value since the value of lSeconds is undefined if the return value is false. For this method to succeed, the csPassword parameter passed to the constructor must be the read/write SNMP community string. This method fails if the csPassword is the read only community string.

See the “[CEtherStack::FindPortFromMac Method](#)” section on page 7-23 and RFC-1493 for a discussion of the transparent bridge forwarding table.

Override this method if your network element sets the aging period for the MAC address to port mapping in a different way. If your network element does not support setting this value, always return false.

```
virtual bool SetAgingTime(long lSeconds);
```

## Parameters

*lSeconds*

The number of seconds that a MAC address entry remains in the forwarding table of the transparent bridge

## CEtherStack::SupportsDefaultPortmap Method

The SupportsDefaultPortmap method returns true if the AddDefaultPortmap method is supported. See the “[CEtherStack::AddDefaultPortmap Method](#)” section on page 7-18.

```
virtual bool SupportsDefaultPortmap() const;
```

There are no parameters for this method.

## CEtherStack::SupportsPortmapUpdate Method

This method is obsolete. It may be removed in a subsequent release.

```
virtual bool SupportsPortmapUpdate() const;
```

There are no parameters for this method.

## CEtherStack::UnpackedMACToOid Method

See the “[CEtherStack::MACToOid Method](#)” section on page 7-29. This method accepts the MAC address in string instead of binary format.

```
static CString UnpackedMACToOid(const CString& csMAC);
```

## Parameters

*csMAC*

The MAC address as string in “xx xx xx xx xx xx” format, where x is a hex digit.

## CEtherStack::UnpackedToMAC Method

This utility takes a MAC address represented as six pairs of hexadecimal digits; that is, “00 10 5a 82 2c fa”, each pair representing a byte, and returns the MAC address in binary format. The method accepts hex digits a–f in both upper and lower case. The BBSM core uses this routine to convert MAC addresses stored in the database to binary form. See the “[CEtherStack::MACtoUnpacked Method](#)” section on page 7-30.

```
static void UnpackedToMAC(MACADDR& MACAddr, const char* szMAC);
```

### Parameters

*MACAddr*

This parameter returns a MAC address in binary form.

*szMAC*

This parameter returns a MAC address in text format.

## CModemStack Class

The CModemStack class, derived from the CEtherStack base class, overrides some methods to serve as a base class for network elements based on modem technology, such as DSL and cable modems. This base class assumes that you will implement AddPortMap to discover network elements and create port map entries for them. It also assumes that the network elements cannot report if a client is connected to them. For such devices, you may prefer to derive your network element interface class from this intermediate base class instead of the CEtherStack class.

There are no member variables for this derived class.

This class provides or overrides the following methods from the CEtherStack base class. Although you can override the specialized methods implemented in this derived class, if you have to do so it may be less work to derive your class from CEtherStack instead.

## CModemStack::CModemStack Method

The CModemStack method calls the constructor for CModemStack from your sub-class constructor. Your constructor is usually called from static “CreateStack” method implemented by your class (see below), which the BBSM core calls to create an instance of your network element module interface object.

As a subclass of CEtherStack, this method calls the CEtherStack constructor.

```
CModemStack  
(  
    const CString& csIP,  
    const CString& csPassword,  
    int nSiteNumber,  
    int nStackNumber = 1  
);
```

### Parameters

*csIP*

IP address of the SNMP agent on the network element.

*csPassword*

SNMP community string (read or read/write) from the network element.

*nSiteNumber*

This number associates the network element with a site in multi-site applications.

*nStackNumber*

Each network element within a site is assigned stack number. This number is the first 4digits of the identifier for a port within a site.

## CModemStack::~CModemStack() Method

The virtual destructor insures that your derived class destructor is called when an object is deleted through a base class pointer.

```
virtual ~CModemStack();
```

There are no parameters for this method.

## CModemStack::AddDefaultPortmap Method

To implement AddPortMap for modem based technologies, you generally need to perform a proprietary modem discovery algorithm and record some information in the port map to allow your class to identify the modem during an end user session. See the “[CEtherStack::AddPortmapEntry Method](#)” section on [page 7-21](#). This implies that you will override the AddPortMap method and that BBSM cannot generate a default port map if the modem head-end is not on line.

For this reason, CModemStack provides an implementation of this method that always returns false. The debug version of this method asserts if it is called.

```
virtual bool AddDefaultPortmap
(
    int                nSwitchCount,
    PINT              pnMinRoomNum,
    const PortMapArgs& pPortMapArgs,
);
```

### Parameters

*nSwitchCount*

Number of switches in stack.

*pnMinRoomNum*

Pointer to room number, incremented for each map entry created.

*pPortMapArgs*

Pointer to a structure of the same parameters described in the “[CEtherStack::AddDefaultPortmap Method](#)” section on [page 7-18](#).

## CModemStack::GetClientPortCount Method

The GetClientPortCount method returns zero and will assert if called in the debugging version.

```
virtual int GetClientPortCount(int nSwitch);
```

### Parameters

*nSwitch*

The switch number within the stack

## CModemStack::GetDisconnectTimeAdjustment Method

This method returns the value from CEtherStack::GetAgingTime, assuming MAC address entries will age out at the head end after this interval. Override this method if your derived class needs to adjust the session duration in a different way.

```
virtual int GetDisconnectTimeAdjustment();
```

There are no parameters for this method.

## CModemStack::SupportsDefaultPortmap Method

This version of the method always returns false.

```
virtual bool SupportsDefaultPortmap() const;
```

There are no parameters for this method.

## CSNMPAgent Class

The CSNMPAgent class communicates with the SNMP agent on the network element. The CEtherStack base class contains a protected member, m\_Agent, which is automatically initialized by the constructor. Use it in your derived classes to get, set and walk objects in the MIB. This section documents the most useful member data and methods of this class.

### Member Variables

`CString m_csIPAddress`

This method contains the IP address of the SNMP agent on the network element. This value comes from the Switches table and is passed in from the network element interface object that contains this object.

## CSNMPAgent::CSNMPAgent Method

The constructor sets the IP address of the agent with which it will communicate and the SNMP community string. The set methods of this class will succeed only if this is the read/write community string. The other methods work with either the read only or the read/write string.

```
CSNMPAgent(const CString& csIPAddress, const CString& csCommunity);
```

## Parameters

*csIPAddress*

The IP address of the SNMP agent accessed by this object.

*csCommunity*

The SNMP community string.

## CSNMPAgent::CloseSession Method

The CloseSession method resets the internal state of the object to allow a subsequent OpenSession. See the Microsoft documentation for SnmpMgrClose.

```
void CloseSession();
```

There are no parameters for this method.

## CSNMPAgent::GetCounter Method

These methods retrieve the value of an object of type ASN\_COUNTER32 from the SNMP agent associated with this object. These methods return true if the agent responds, the object exists and is of the correct type. You must call OpenSession before you call these methods. See the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier.

```
bool GetCounter(AsnObjectIdentifier& csOID, unsigned long& value);
```

```
bool GetCounter(const char* oidString, unsigned long& value);
```

## Parameters

*csOID*

The object identifier for the object of type ASN\_COUNTER32 (AsnObjectIdentifier) to retrieve.

*oidString*

The object identifier for the object of type ASN\_COUNTER32 (string).

*value*

If the method returns true, the counter value returned by the agent.

## CSNMPAgent::GetInteger Method

These methods retrieve the value of an object of type ASN\_INTEGER from the SNMP agent associated with this object. These methods return true if the agent responds, the object exists and is of the correct type. You must call OpenSession before you call these methods. See the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier.

```
bool GetInteger(AsnObjectIdentifier& csOID, long& value);
```

```
bool GetInteger(const char* oidString, long& value);
```

## Parameters

*csOID*

The object identifier for the object of type ASN\_INTEGER (AsnObjectIdentifier) to retrieve.

*oidString*

The object identifier for the object of type ASN\_INTEGER to retrieve.

*value*

If the method returns true, the integer value returned by the agent.

## CSNMPAgent::GetIPAddress Method

The GetIPAddress method retrieves the value of an object of type ASN\_IPADDRESS from the SNMP agent associated with this object. This method returns true if the agent responds, the object exists and is of the correct type. You must call OpenSession before you call this method. See the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier.

```
bool GetIPAddress(const char* oidString, CString& value);
```

### Parameters

*oidString*

The object identifier for the object of type ASN\_IPADDRESS to retrieve.

*value*

If the method returns true, the IP address value returned by the agent.

## CSNMPAgent::GetNextInteger Method

The object identifiers in a MIB supply a tree. To walk a portion of this tree, call StartWalk with the root of the oid tree. Each call of this method retrieves the next object with the same identifier prefix. This method returns true if the agent responds, another object exists with the same prefix and is of the correct type. You must call OpenSession and StartWalk before you call this method. See the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier.

```
bool GetNextInteger(CString &oid, long& value);
```

### Parameters

*oid*

Returns the object identifier for the object retrieved.

*value*

If the method returns true, the integer value returned by the agent.

## CSNMPAgent::GetNextIPAddress Method

The object identifiers in a MIB supply a tree. To walk a portion of this tree, call StartWalk with the root of the oid tree. Each call of this method retrieves the next object with the same identifier prefix. This method returns true if the agent responds, another object exists with the same prefix and is of the correct type. You must call OpenSession and StartWalk before you call this method. Refer to the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier.

```
bool GetNextIPAddress(CString &oid, CString &value);
```

## Parameters

*oid*

Returns the object identifier for the object retrieved.

*value*

If the method returns true, the ASN\_IPADDRESS value returned by the agent.

## CSNMPAgent::GetNextObjectIdentifier Method

The object identifiers in a MIB supply a tree. To walk a portion of this tree, call StartWalk with the root of the oid tree. Each call of this method retrieves the next object with the same identifier prefix. This method returns true if the agent responds, another object exists with the same prefix and is of the correct type. You must call OpenSession and StartWalk before you call this method. See the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier.

```
bool GetNextObjectIdentifier(CString& oid, CString& value);
```

## Parameters

*oid*

Returns the object identifier for the object retrieved.

*value*

If the method returns true, the ASN\_OBJECTIDENTIFIER value returned by the agent.

## CSNMPAgent::GetNextOctetString Method

The object identifiers in a MIB supply a tree. To walk a portion of this tree, call StartWalk with the root of the tree oid. Each call of this method retrieves the next object with the same identifier prefix. This method returns true if the agent responds, another object exists with the same prefix and is of the correct type. You must call OpenSession and StartWalk before you call this method. See the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier.



### Note

SMNP agents represent MAC addresses as 6-byte octet strings.

```
bool GetNextOctetString(CString& oid, PBYTE pbyData, size_t length);
```

## Parameters

*oid*

Returns the object identifier for the object retrieved.

*value*

If the method returns true, the ASN\_OCTETSTRING value returned by the agent.

## CSNMPAgent::GetObjectIdentifier Method

The `GetObjectIdentifier` method retrieves the value of an object of type `OBJECTIDENTIFIER` from the SNMP agent associated with this object. This method returns true if the agent responds, the object exists and is of the correct type. You must call `OpenSession` before you call this method. See the Microsoft documentation for `SnmpMgrRequest` and `AsnObjectIdentifier`.

```
bool GetObjectIdentifier (const char* oidString, CString& value);
```

### Parameters

*oidString*

The object identifier for the object of type `ASN_OBJECTIDENTIFIER` to retrieve.

*value*

If the method returns true, the `ASN_OBJECTIDENTIFIER` value returned by the agent.

## CSNMPAgent::GetOctetString Method

These methods retrieve the value of an object of type `ASN_OCTETSTRING` from the SNMP agent associated with this object. These methods return true if the agent responds, the object exists and is of the correct type. You must call `OpenSession` before you call these methods. See the Microsoft documentation for `SnmpMgrRequest` and `AsnObjectIdentifier`.



### Note

---

SNMP agents represent MAC addresses as 6-byte octet strings.

---

```
bool GetOctetString (AsnObjectIdentifier & oid, AsnOctetString& value);
```

```
bool GetOctetString (const char* szOid, PBYTE pbyData, size_t length);
```

### Parameters

*csOID*

The object identifier for the object of type `ASN_OCTETSTRING` to retrieve.

*oidString*

The object identifier for the object of type `ASN_OCTETSTRING` to retrieve.

*value*

If the method returns true, the octet string value returned by the agent.

## CSNMPPAgent::OpenSession Method

Call the `OpenSession` method before attempting to access the agent's MIB through the `StartWalk`, `GetXXX`, and `SetXXX` methods. This method initializes the interface to the MIB. See the Microsoft documentation for the `SnmpMgrOpen` routine.



### Note

The retry interval and the number of retries are controlled by the `TimeOut` and `Retries` `REG_DWORD` values under the registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IORT\SNMP
The default values are 6000 milliseconds and 3 retries.
```

```
bool OpenSession();
```

There are no parameters for this method.

## CSNMPPAgent::SetInteger Method

These methods set the value of an object of type `ASN_INTEGER` in MIB of the SNMP agent associated with this object. These methods return true if the agent responds, the object exists, is of the correct type and is successfully set. You must call `OpenSession` before you call these methods. See the Microsoft documentation for `SnmpMgrRequest` and `AsnObjectIdentifier`. These methods will fail if the read only community string was passed to the constructor.

```
bool SetInteger(AsnObjectIdentifier& oid, long lValue);
```

```
bool SetInteger(const char* oidstring, long lValue);
```

### Parameters

*oid*

The object identifier for the object of type `ASN_INTEGER` (`AsnObjectIdentifier`) to set.

*oidString*

The object identifier for the object of type `ASN_INTEGER` to set.

*value*

The integer value to be set by the agent.

## CSNMPPAgent::SetIPAddress Method

These methods set the value of an object of type `ASN_IPADDRESS` in MIB of the SNMP agent associated with this object. These methods return true if the agent responds, the object exists, is of the correct type and is successfully set. You must call `OpenSession` before you call these methods. See the Microsoft documentation for `SnmpMgrRequest` and `AsnObjectIdentifier`. These methods will fail if the read-only community string was passed to the constructor.

```
bool SetIPAddress(AsnObjectIdentifier &oid, AsnOctetString &string);
```

```
bool SetIPAddress(const char* oidstring, const CString &csIPAddress);
```

## Parameters

*oid*

The object identifier for the object of type ASN\_IPADDRESS (AsnObjectIdentifier) to set.

*oidString*

The object identifier for the object of type ASN\_IPADDRESS to set.

*value*

The IPADDRESS value to be set by the agent.

## CSNMPAgent::SetOctetString Method

The SetOctetString method sets the value of an object of type ASN\_OCTETSTRING in MIB of the SNMP agent associated with this object. This method returns true if the agent responds, the object exists, is of the correct type and is successfully set. You must call OpenSession before you call these methods. Refer to the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier. This method will fail if the read only community string was passed to the constructor.



Note

SNMP agents represent MAC addresses as 6-byte octet strings.

```
bool SetOctetString(const char* szOid, const unsigned char* pbyData,
                   size_t length);
```

## Parameters

*szOid*

Object identifier of the ASN\_OCTETSTRING to set in the SNMP agent's MIB.

*pbyData*

Pointer to the octet string to set.

*length*

The length of the octet string to set.

## CSNMPAgent::StartWalk Method

The object identifiers in a MIB supply a tree. To walk a portion of this tree, call StartWalk with the root of the oid tree. Each call of a GetNext method retrieves the next object with the same identifier prefix. This method returns true if the object successfully allocates the internal data structures required to implement the MIB walk. You must call OpenSession before you call this method. See the Microsoft documentation for SnmpMgrRequest and AsnObjectIdentifier.

```
bool StartWalk(const char* oidStringRoot);
```

## Parameters

*oidStringRoot*

The prefix of the identifiers of objects to retrieve

