NerveCenter 6.0 AddVarbinds API

Windows and UNIX
Version 6.0

August 2012 NCAVB60-01

Contacting LogMatrix
LogMatrix, Inc.
4 Mount Royal Ave, Suite 250
Marlborough, MA 01752

Phone 508-597-5300
Fax 774-348-4953

contact email: info@logmatrix.com

Website: www.logmatrix.com
Forum: http://community.logmatrix.com/LogMatrix
Blog: www.logmatrix.com/blog

NerveCenter AddVarbinds API

The AddVarbinds API, introduced to NerveCenter with the NC5.2 and NC6.0 releases, provides the ability for enhancing the data being processed by Alarms, Poll Functions and Trap Masks. Through this API, data retrieved by polls and data received from event notifications can be enriched and passed on to other elements of your site's management solution.

In Part One, that follows immediately, this document presents three usage examples for the AddVarbinds API and then in Part Two provides a reference guide on the API's functions.

The AddVarbinds API is available in the same format across NerveCenter's supported platforms (Linux, Solaris and Windows). The API is included as part of the NC5.2 and NC6.0 releases and does not require further component acquisition or download.

If your site is using versions of NerveCenter prior to NC6.0, contact your NerveCenter sales representative and enquire on how your NerveCenter systems can be upgraded.

Part One: Working with the AddVarbinds API

First Example: Enhancing the common LinkDown SNMP Trap

Network devices such as servers, routers, switches, workstations and even notebooks, when equipped with an SNMP Agent, give off the standard set of SNMP Traps. LinkUp, LinkDown, ColdStart, AuthenticationFailure are the commonly seen traps these devices give off to your network management infrastructure.

Running NerveCenter's **nclistener** in a terminal window might show the live receipt of these kinds of notifications. Here is an example of nclistener running on a workstation named nervecenter.

```
nervecenter{gmoberg}501: /opt/OSInc/bin/nclistener
nclistener Version 6.0.02(buildID 6002), Copyright (C) 1989-2012 LogMatrix Inc.

Listening for SNMP Notifications

2012-06-21 14:32:49
SNMPv2c Trap { from(10.1.50.202:4) trapOID(linkDown) ticks(870550781) community(public)}
      #vb1: {ifIndex.75497818, 75497818/Integer32}
      #vb2: {ifAdminStatus.75497818, 1/Integer32}
      #vb3: {ifOperStatus.75497818, 2/Integer32}

2012-06-21 15:22:08
SNMPv2c Trap { from(10.1.50.202:4) trapOID(linkUp) ticks(870846834) community(public)}
      #vb1: {ifIndex.75497818, 75497818/Integer32}
      #vb2: {ifAdminStatus.75497818, 1/Integer32}
      #vb3: {ifOperStatus.75497818, 1/Integer32}
```

Here, one LinkDown and one LinkUp notification is received by the management station. The content of these two traps matches the definition for each as found in the IF-MIB module of RFC2863. Each notification reports the alert itself – LinkUp or LinkDown – and then three pieces of data pertaining to this occurrence of the alert.

Here are the definitions from RFC2863:

```
-- definition of interface-related traps.

linkDown NOTIFICATION-TYPE
   OBJECTS { ifIndex, ifAdminStatus, ifOperStatus }
   STATUS  current
   DESCRIPTION
          "A linkDown trap signifies that the SNMP entity, acting in
          an agent role, has detected that the ifOperStatus object for
          one of its communication links is about to enter the down
          state from some other state (but not from the notPresent
          state).  This other state is indicated by the included value
          of ifOperStatus."
   ::= { snmpTraps 3 }

linkUp NOTIFICATION-TYPE
   OBJECTS { ifIndex, ifAdminStatus, ifOperStatus }
   STATUS  current
   DESCRIPTION
          "A linkUp trap signifies that the SNMP entity, acting in an
          agent role, has detected that the ifOperStatus object for
          one of its communication links left the down state and
          transitioned into some other state (but not into the
          notPresent state).  This other state is indicated by the
          included value of ifOperStatus."
   ::= { snmpTraps 4 }
```

Catching these alert notifications in NerveCenter is easy using its Trap Mask functionality. The NerveCenter sample database contains simple Trap Mask examples for these two events.

But, let's say your processes need to add information to such an event. Perhaps the requirement is that the network management infrastructure needs to record these events, including the name of the reporting device, a formatted timestamp and the name of NerveCenter server receiving the notification. Including items such as these is where the AddVarbinds API is really put to use.

To implement the solution for this requirement, the first step is to add the three values to the trap data when a linkUp or linkDown trap is received. In the Trap Mask dialog for the linkDown event type, the following lines of Perl code could be used.

```
# TrapMask Function for LinkDown.

# Return a timestamp in "yyyy-mm-dd day hh:mm:ss" format
sub timestamp {
   my ( $second, $minute, $hour, $day, $month, $year, $weekday, $dayOfYear, $daylightSavings) = localtime();
   sprintf( "%04d-%02d-%02d %02d:%02d:%02d", $year+1900, $month+1, $day, $hour, $minute, $second );
}

# Add varbinds to the notification's data payload.
AddVarbind( $NodeName );       # Add the name of the node reporting the event.
AddVarbind( $NCHostName );     # Add the name of this management station.
AddVarbind( &timestamp );      # Add the current date & time.

# Fire alert trigger
FireTrigger( "linkDown" );
```

The first part, a subroutine, is to generate the timestamp. It formats the time, as reported by Perl's localtime() function, into the required format.

Next comes three lines where the AddVarbind() function is called. These three lines add the three required data elements, one at a time. $NodeName, $NCHostName are built-in values, automatically provided by NerveCenter when a Trap Mask Function is called. And the third call to AddVarbind() is set to first call the timestamp() function, thereby having AddVarbind() receive the value returned – the formatted date & time.

You could repeat this for the Trap Mask that handles LinkUp notifications. The same Perl code would be used.

The next step is to create an Alarm, where "linkDown" and "linkUp" trigger events are used to drive state transitions. The Alarm's transitions can be used to test our AddVarbind work by adding actions where the augmented data is re-emitted as an output from NerveCenter.

Since we're already talking about Traps, let's use the 'Send Trap' action. Setting up the 'Send Trap' action to issue the trap back to localhost allows the same nclistener to display our modifications. However, it is important to change the name of the trap from linkDown to something else. Otherwise an infinite loop situation would be set up.

Here is the output from nclistener upon the occurrence of the next linkDown event. The first event is the incoming linkDown notification. The second is the result of the re-packaging performed by the above Perl logic and the issuing with a 'Send Trap' action. The re-packaged notification has been sent as an enterprise-specific trap, number #111. This avoids sending a redundant linkDown event back to our NerveCenter Server.

```
2012-06-22 13:41:21
SNMPv2c Trap { from(10.1.50.202:4) trapOID(linkDown) ticks(878882203) community(public)}
    #vb1: {ifIndex.75497794, 75497794/Integer32}
    #vb2: {ifAdminStatus.75497794, 1/Integer32}
    #vb3: {ifOperStatus.75497794, 2/Integer32}

2012-06-22 13:41:24
SNMPv1 Trap { from(10.1.50.107:44984) agent(10.1.50.202) generic(6 "enterpriseSpecific") specific(111) enterprise(openservice)
ticks(764863027) community(public)}
    #vb1: {ifIndex.75497794, 75497794/INTEGER}
    #vb2: {ifAdminStatus.75497794, 1/INTEGER}
    #vb3: {ifOperStatus.75497794, 2/INTEGER}
    #vb4: {osincVarbindValue.1, 3Com-SS3-4500-B/OCTET STRING[15]}
    #vb5: {osincVarbindValue.2, nervecenter/OCTET STRING[5]}
    #vb6: {osincVarbindValue.3, 2012-6-22 13:41:22/OCTET STRING[18]}
```

The first event above is a linkDown notification from a 3Com SuperStack3 switch at 10.1.50.202. Our Trap Mask handler for linkDown events is called, adds its three varbinds and then fires a 'linkDown' trigger. This causes an Alarm we have prepared to transition and call a 'Send Trap' action. This dutifully emits the event and, as the nclistener output shows, the data now contains our AddVarbind values in addition to the original notification's data payload. "3Com-SS3-4500-B" is the name of the device at 10.1.50.202 (aka "$NodeName"), "nervecenter" is the name of the management station (aka "$NCHostName"), and the time has been added using our required format.

Working with Varbinds

As this first example has shown, SNMP provides data as part of its event notifications. The data contained in the event is provided as an orderly arranged set of items. The order and arrangement of the items is specified by the rules of SNMP and by the definition of the event within an SNMP MIB module. The definition, for example, of the linkDown trap notification specifies three pieces of data are to be included. The ordering given for such event definitions, exemplified by the linkDown definition in IF-MIB, also infers the data's ordering when the event occurs.

SNMP calls these data items Variable Bindings, or *varbinds* for short. To refer again to our first example, the linkDown event is defined to contain three varbinds: ifIndex, ifAdminStatus and ifOperStatus. Our use of the AddVarbinds API grew a linkDown notification's initial set of three to six varbinds.

Although we have not yet seen an example of SNMP polling, it too uses the organization of arranging a message's contained data into varbinds. And as we shall see in the next example, the AddVarbinds API can be used for enriching poll responses, just as it was used above for enriching a notification event's data.

What is a varbind?

SNMP operates by very specific rules, including how it packages data. A varbind is the packaging of a single data item. The packaging includes more than just the value. Every varbind is a set of three items: a name, a data type indicator, and the value. If you return to the nclistener output in our first example, you'll find these per-varbind elements. Note how each varbind, such as "ifOperStatus.75497794, 2/INTEGER" or "osincVarbindValue.2, nervecenter/OCTET STRING[5]", contains these elements. Specifically,

- "ifOperStatus.75497794" and "osincVarBindValue.2" are names,
- "INTEGER" and "OCTET STRING[5]" are type labels, and
- "2" and "nervecenter" are values.

Let's look at what SNMP means by a varbind's name and data type.

SNMP Naming

SNMP uses object identifiers, OIDs, as names. OIDs are numeric sequences, commonly expressed textually as a sequence of numbers where each is separated by a period. "1", "100", "1.2", "100.300", "1.3.5.7.0.9" are all examples of OIDs. To be valid, an OID must be at least one value in length and each element of the OID sequence must be a non-negative integer. Thus "" (an empty value), "-100", "1.2.3.abc" and "1.-33.8" are not valid OIDs.

Because OIDs are cumbersome to read and write they are often replaced with recognizable names. MIB modules are the means whereby OIDs are defined and given a human-readable name. In SNMP, names are to start with a lowercase letter and then may contain any number of upper or lower case letters, numbers and hyphens.

Here are some example OIDs and their textual name.

Table 1 Sample OID-to-Name mappings

OID	Name	Defined in
1.3.6.1.2.1.1	system	SNMPv2-MIB
1.3.6.1.2.1.1.5	sysName	SNMPv2-MIB
1.3.6.1.2.1.2.2.1.2	ifDescr	IF-MIB
1.3.6.1.2.1.17.1.4	dot1dBasePortTable	BRIDGE-MIB
1.3.6.1.6.3.1.1.5.3	linkDown	IF-MIB

Most tools hide OIDs and replace them with their textual names. Many parts of NerveCenter do this.

Running nclistener in two terminal session windows at the same time can show the replacement it is doing. Running "nclistener –oids" indicates that OIDs should be shown. Running without "-oids" indicates the OIDs should be replaced with their textual names.

Here is an example of two ncicmplistener applications reporting the same SNMPv2c linkDown trap. Note how the second nclistener shows the OIDs for the varbinds and the TrapOID value whereas the first nclistener translate these items to text names.

```
[nervectr@megan ~]$ /opt/OSInc/bin/nclistener

nclistener Version 6.0.02(buildID 6002 BLD11), Copyright (C) 1989-2012 LogMatrix Inc.

Listening for SNMP Notifications

2012-06-25 15:49:15
SNMPv2c Trap { from(10.1.50.202:4) trapOID(linkDown) ticks(905695918) community(public)}
      #vb1: {ifIndex.75497578, 75497578/Integer32}
      #vb2: {ifAdminStatus.75497578, 1/Integer32}
      #vb3: {ifOperStatus.75497578, 2/Integer32}
```

```
[nervectr@megan ~]$ /opt/OSInc/bin/nclistener -oids

nclistener Version 6.0.02(buildID 6002 BLD11), Copyright (C) 1989-2012 LogMatrix Inc.

Listening for SNMP Notifications

2012-06-25 15:49:15
SNMPv2c Trap { from(10.1.50.202:4) trapOID(1.3.6.1.6.3.1.1.5.3) ticks(905695918) community(public)}
      #vb1: {1.3.6.1.2.1.2.2.1.1.75497578, 75497578/Integer32}
      #vb2: {1.3.6.1.2.1.2.2.1.7.75497578, 1/Integer32}
      #vb3: {1.3.6.1.2.1.2.2.1.8.75497578, 2/Integer32}
```

The same OID tagging and translation action applies to polling. Requests made to an SNMP Agent specify what data is looked up and returned. To do this, the request specifies an OID. When the SNMP Agent replies, it packages the requested data into a varbind, naming the object's OID, type and value.

Here are two lookups of the MIB object "sysName". Both are performed with NerveCenter's **ncget** utility. The first shows the name translation, as usual. The second uses the "-oids" parameter and thus shows the raw OIDs.

```
[nervectr@nervectr ~]$ ncget -v2c 10.1.50.201 public sysName.0
Sending to 10.1.50.201 161/udp
Normal SNMPv2c reply
     #vb1: {sysName.0, 3Com-SS3-4500-A/OCTET STRING[15]}
```

```
[nervectr@nervectr ~]$ ncget -oids -v2c 10.1.50.201 public sysName.0
Sending to 10.1.50.201 161/udp
Normal SNMPv2c reply
     #vb1: {1.3.6.1.2.1.1.5.0, 3Com-SS3-4500-A/OCTET STRING[15]}
```

Automatic OID Generation

Recognizing that OIDs are onerous to deal with, the AddVarbinds API has been set up to allow for automatic OID generation. While the API permits that OIDs can be specified as needed, it optionally allows for them to be omitted from calls. The result of omitting an OID is an auto generation of one quietly within the API's implementation.

The MIB object **osincVarbindValue** is used as the base name for automatic OID generation. (This name has the OID translation "1.3.6.1.4.1.78.11.1.1.10.1.1.2"). Whenever a varbind is added via the API and no OID is named, this object is used as the base and given an incremental value as its suffix. For example, if the API is used to add the three varbinds "one", "two" and "three" without naming any OID, the first will be automatically assigned the name "osincVarbindValue.1", the second "osincVarbindValue.2" and the third "osincVarbindValue.3".

Thus the Perl code

```
AddVarbind( "one" );
AddVarbind( "two" );
AddVarbind( "three" );
```

could lead to the following, if this augmenting of a SNMP trap was re-transmitted as a SNMP Trap

```
     #vb1: {osincVarbindValue.1, one/OCTET STRING[3]}
     #vb2: {osincVarbindValue.2, two/OCTET STRING[3]}
     #vb3: {osincVarbindValue.3, three/OCTET STRING[5]}
```

This automatic OID generation is restarted for every trap occurrence or poll response. The first added varbind, whether this occurs for a Poll Function, Trap Mask Function or a Perl Subroutine Action, where the auto generation is used, will start anew with osincVarbindValue.1 and move to .2 and then .3 with each successive call to the AddVarbinds API.

SNMP Data Types

The second part of the varbind packaging is the data type identification.

SNMP defines a set of data types. The original version of SNMP (ca. 1990) introduced INTEGER, Counter, Gauge, OctetString, Object Identifier, and Opaque. SNMP v2 (ca. 1994) added Counter64, Unsigned32 and improved the definition of TEXTUAL-CONVENTION. SNMP v2 also provided replacements for several of v1's types: Counter became Counter32, Gauge became Gauge32, and INTEGER is matched with Integer32. (SNMPv3 supports the same set as v2 without change.)

These data types are used for describing the information conveyed using SNMP. For example, if a message contains data about how quickly an interface on a device is receiving traffic, the data for that rate would arrive accompanied with a data type identifier. This allows the message receiver to understand how to interpret the associated value. In this example, the data type identifier would likely be "Integer32". As another example, if the hostname of a device is retrieved over SNMP, the returned value (ex: "SalesDeptPrinter4") would likely be defined as being an "Octet String".

Table 2 shows the set of SNMP types and their history. The "Data Types" column names each defined data type. The "SNMP Version" columns show which of SNMPs three versions supports each data type. The "ASN.1 BER Tag" column shows the encoding value used for type identification when an SNMP message is sent on the network.

Note that the SNMP version columns are for historical reference only. Your use of the AddVarbinds API does not require adherence to this table. For example AddCounter64Varbind can be invoked for augmenting the return from an SNMP v1 poll or a v1 trap as well as a v2 or v3 poll or trap.

Table 2 SNMP Data Types

Data Types	SNMP Version			ASN.1 BER Tag	Note
	v1 RFC1155 (Official)	v2 RFC1442 (Historic)	v2 and v3 RFC2578 (Official)		
INTEGER	Yes	Yes	Yes	0x02	Signed 32-bit integer, Enumerations
Integer32	No	Yes	Yes		Signed 32-bit integer
OCTET STRING	Yes	Yes	Yes	0x04	
BITS	No	No	Yes		
OBJECT IDENTIFIER	Yes	Yes	Yes	0x06	
IpAddress	Yes	Yes	Yes	0x40	IPv4 only
Counter	Yes	Yes	Allowed	0x41	Non-negative, non-decreasing 32-bit
Counter32	No	No	Yes		
Gauge	Yes	Yes	Allowed	0x42	Non-negative 32-bit values.
Gauge32	No	Yes	Yes		
Unsigned32	No	No	Yes		
TimeTicks	Yes	Yes	Yes	0x43	Non-negative, non-decreasing 32-bit values
Opaque	Yes	Yes	Deprecated, Allowed	0x44	SMIv1 MIB Modules only
NsapAddress	No	Yes	No	0x45	Deprecated
Counter64	No	Yes	Yes	0x46	Non-negative, non-decreasing 64-bit values
UInteger32	No	Yes	No	0x47	Deprecated
BIT STRING	No	Yes	No	?	Deprecated

A second example: Enhancing Poll Responses

TBD

A third example: Enhancing Alarm-level data

TBD

Part Two: The AddVarbinds API Reference

AddVarbind

```
AddVarbind( $value );
        AddVarbind( “Printer” );
        AddVarbind( “Connected @ 12:30:15 08/31/2012” );
AddVarbind( $oid, $type, $value );
        AddVarbind( “1.3.6.1.2.1.2.2.1.13”,” integer”, 110 );
```

The base call of the API, **AddVarbind()**, can be used in either of two manners. The two can be used interchangeably and repeatedly by a Poll Function, Trap Mask Function or Perl Subroutine Action.

AddVarbind() appends the provided ***$value*** as an additional varbind to the current varbind set, labeling it with the an ***$oid*** value and identifying the value’s data type according to a value assigned to ***$type***. In the first format, ***$oid*** is automatically created and ***$type*** is set to ***“string”***.

AddVarbind($value)

The first format, ***AddVarbind($value)***, accepts a single parameter, which is the always accepted as a character string. ***$value*** needs to be defined. This form of AddVarbind(), then, is equivalent to AddStringVarbind($value). The OID for this form is automatically created as described in the Automatic OID Generation section.

Defined and undefined variables:

The Perl language supports that variables can be defined or not defined. Writing “my $name;” declares “$name” as a scalar variable; but no value has been assigned. Thus, $name is *undefined*.

In order to define a variable, it needs to be assigned to a value. “my $name = ‘fred’;” or “$name = ‘roger’;” are examples of defining the variable $name.

To set a variable to the *undefined* state, either don’t give it a value or else use the ‘undef’ keyword. For example “undef $name;” forces $name back to be undefined. Also “$name = undef;” will set $name to be undefined.

The keyword “defined” can be used to test whether a variable is defined or not defined. For example “print ‘My name is $name.\n” if defined $name;” will print “My name is joe.” if $name is set to “joe” but will not print anything if $name is not defined.

Throughout the AddVarbind API, it is important to pass only defined values to the API’s functions. If you write “my $system; AddVarbind($system);”, the call will detect the undefined value and return without adding anything.

AddVarbind($oid, $type, $value)

The second form, ***AddVarbind($oid, $type, $value)***, requires three defined parameters. This format provides the most flexibility of all the calls in the API. All the other calls, except for TraceAddVarbinds, map to this call, each providing one or more simplifications. Thus, this call is the least likely to be used yet provides the broadest range of possible uses.

Example #1 Usage of AddVarbind($oid, $type, $value)

```
AddVarbind( "1.2.3", "string", "east coast domain" );

my $rate = delta( ifEntry.ifInOctets ) / $elapsed;
AddVarbind( "1.3.19.20", "integer", $rate );
```

The value of ***$oid*** needs to be an object identifier (ex: "1.3.6.1.2.1.1.1.0"). The value assigned to ***$oid*** can be any validly formed OID (ex: not "1.a.2.3" [the 'a' is invalid, each element of the OID sequence must be a non-negative integer]). The OID does not need to be known to the NerveCenter MIB. OIDs can be either statically defined or else generated at runtime. An empty ***$oid*** value or an undefined ***$oid*** similarly result in the function terminating without performing any action.

> The requirement of the $oid parameter defined above is universal in the AddVarbinds API. Thus, the requirement is also valid for AddIntegerVarbind($oid, $value), AddStringVarbind($oid, $value), AddGaugeVarbind($oid, $value) and any other likewise equivalent calls.

Example #2 Usage examples of $oid parameter for AddVarbind()

```
my $value = 6;
my $type = "integer";

my $oid = undef;
AddVarbind( $oid, $type, $value );  # This will fail because $oid is undefined

$oid = "";
AddVarbind( $oid, $type, $value );  # This will fail because $oid is an empty value

$oid = "abc";
AddVarbind( $oid, $type, $value );  # This will fail because $oid is not a value OID

$oid = "100.200";
AddVarbind( $oid, $type, $value );  # Successful usage
```

The ***$type*** value identifies the SNMP Data type. This value needs to be one of the following strings: "integer", "string", "oid", "ipaddress", "counter", "gauge", "counter64", or "unsigned" as shown in Table 3. An empty value or a value outside of this range will result in the function returning without performing any operation.

Table 3 AddVarbinds API Data Type naming

Data Types	SNMP			AddVarbinds API naming
	v1 RFC1155	FC1442 (Historic)	v2/v3 RFC2578	
INTEGER	Yes	Yes	Yes	“integer”
Integer32	No	Yes	Yes	
OCTET STRING	Yes	Yes	Yes	“string”
BITS	No	No	Yes	
OBJECT IDENTIFIER	Yes	Ye	Yes	“oid”
IpAddress	Yes	Yes	Yes	“ipaddress”
Counter	Yes	Yes	Allowed	“counter”
Counter32	No	No	Yes	
Gauge	Yes	Yes	Allowed	“gauge” or “unsigned”
Gauge32	No	Yes	Yes	
Unsigned32	No	No	Yes	
TimeTicks	Yes	Yes	Yes	“timeticks”
Opaque	Yes	Yes	Deprecated, Allowed	“string”
NsapAddress	No	Yes	No	-
Counter64	No	Yes	Yes	“counter64”
UInteger32	No	Yes	No	-
BIT STRING	No	Yes	No	-

The ***$value*** parameter needs to be set to a value that is appropriate for the named ***$type*** value. For example a ***$value*** of “Marketing Dept.” would not work if the setting of ***$type*** is “integer”.

AddVarbinds

```
AddVarbinds( @values );

        AddVarbinds( “Gateway”, “Connection Lost”, $NodeName, $IPAddress );



        my @items = ( ‘link failure’, $NodeName, ‘priority’ );

        AddVarbinds( @items );



        AddVarbinds( ( ‘service’, ‘customer interface’, ‘traffic limit exceeded’ ) );
```

AddVarbinds() provides a means of quickly adding a collection of values to a varbind set. Each element in the list is added as an individual varbind. The additions are performed in the order provided.

Each of the elements of the list must be a defined value.

The implementation of **AddVarbinds()** is simply a loop that calls **AddVarbind($value)** for each item in given Perl list variable.

AddIntegerVarbind

```
AddIntegerVarbind( $value )

        AddIntegerVarbind( 5 );

        AddIntegerVarbind( $elapsed );

AddIntegerVarbind( $oid, $value )

        AddIntegerVarbind( “1.2.3.4.5”,  100 );

        AddIntegerVarbind( “1.2.3.4.6” , -11234 );
```

AddIntegerVarbind() adds ***$value*** as a varbind to the current varbind set as an integer and labeled with the given ***$oid***. In the first form of the call, the varbind is added using an OID that is automatically generated by NerveCenter.

The value contained in $value needs to be defined and set to an integer value.

Due to Perl’s weak data type mechanism, ***$value*** can be set to a string (ex: my $rate = “33”;) or a number (ex: my $rate = 33;)

Examples of **AddIntegerVarbind**:

```
AddIntegerVarbind( 0 );

my $rate = 33;
AddIntegerVarbind( “100”, $rate );

$rate = “33”;
AddIntegerVarbind( $rate );

AddIntegerVarbind( “10.20.30”, “100” );
AddIntegerVarbind( “1.3.6.1.4.1.78.0.100”, “0” );
AddInteger Varbind(  “-100” );
```

AddIntegerVarbind($value)

Appends a varbind to the current set where $value represents a whole number. The varbind will be of type “integer” and will be set to the current contents of $value. $value needs to be defined and $value should be set to a whole number.

AddIntegerVarbind($oid, $value)

Appends a varbind to the current set, where $oid names the object-identifier to be used for the new varbind. $value needs to be set to a whole number. $oid needs to be set to a object-identifier.

This call is the equivalent of AddVarbind($oid, “integer”, $value);

AddStringVarbind

```
AddStringVarbind( $value )

          AddStringVarbind( “Printer Paper Alert” );

          my $tray = “Lower paper drawer”;

          AddStringVarbind( $tray );

AddStringVarbind( $oid, $value )

          AddStringVarbind( “1.2.3.4.5”,  “Lobby Conference Room” );

          AddStringVarbind( “1.2.3.4.6” , $NodeName );
```

The AddStringVarbind() functions append the provided $value to the current varbind set, labeling the addition as an OCTET STRING – one of SNMP’s data types.

The contents of $value can be any valid Perl string. This includes the empty string, “” (a string of zero length, as produced with ‘my $name = “”; ‘), but does not include an undefined value. Thus “my $name = undef;” creates a situation where passing $name to AddStringVarbind() would case the function to fail.

AddStringVarbind($value)

Appends a new varbind to the current varbind set, automatically creating the new varbind’s OID and setting the type to “string”. The call is the equivalent of AddVarbind($value) ; both of these calls add the varbind, accepting $value as a character string.

AddStringVarbind($oid, $value)

Appends a new varbind to the current varbind set, using $oid as the OID value and the contents of $value as the varbind’s value – taken to be a character string. The call is the equivalent of AddVarbind($oid, “string”, $value).

AddGaugeVarbind

```
AddGaugeVarbind( $value )

        AddGaugeVarbind( 100 );


        my $temperature = 300;

        AddStringVarbind( “Kelvin” );

        AddGaugeVarbind( $temperature );

        $temperature = $temperature – 273 ;

        AddStringVarbind( “Celsius” );

        AddGaugeVarbind( $temperature );

        $temperature = ( 5 / 9 ) * $temperature + 32 ;

        AddStringVarbind( “Fahrenheit” );

        AddGuageVarbind( $temperature );


AddGaugeVarbind( $oid, $value )

        AddGaugeVarbind( “1.2.3.4.5”,  1000000 );
```

AddGaugeVarbind, AddCounterVarbind and AddUnsignedVarbind are a family of calls that work identically. In each case the value assigned into the varbind is an unsigned, 32-bit integer. The range, then, is 0 up to 4,294,967,295, inclusive.

The role of SNMP’s Gauge data type, as opposed the Counter and Unsigned data types, is to represent an object where the object’s value should move between a high and low. The value of a gasoline engine’s fuel tank moves between ‘F’ and ‘E’ for example. The Kelvin temperature scale is a good second example.

Counter, by comparison, is to represent a value that only increases. The odometer of a car is a suitable example.   Counters can only increase in value or be reset to zero.

Unsigned, the third of these SNMP data types, simply represents an arbitrary value where the range is constrained to non-negative whole numbers.

An oddity of SNMP is that the Gauge, Gauge32, and Unsigned32 data types share the same encoding mark. The Basic Encoding Rules (BER) used by SNMP tag these data types with the value 0x42 (hex), as shown in Table 2. Therefore a management application receiving varbinds of these types are left to guess which data type (Gauge, Gauge32, Unsigned32) was intended.

AddGaugeVarbind($value)

Appends a new varbind to the current varbind set, where the contents of $value are expected to represent that of an unsigned, 32-bit integer. The OID for the added varbind is automatically generated by the API and the type value is set to “gauge”.

AddGaugeVarbind($oid, $value)

Appends a new varbind to the current varbind set, where the contents of $value are expected to represent that of an unsigned, 32-bit integer and the OID for varbind will be set according to the contents of $oid. The varbind’s type will be set to “gauge”. The call is the equivalent of “AddVarbind($oid, “gauge”, $value);” .

AddCounterVarbind

```
AddCounterVarbind( $value )

        AddCounterVarbind( 1843 );



        my $allowance  = ifEntry.ifInOctets + 10000;

        AddCounterVarbind( $allowance );



AddCounterVarbind( $oid, $value )

        AddCounterVarbind( “1.3.6.1.4.1.78.3.4.5”,  $numTrapsReceived  );
```

The two AddCounterVarbind() functions append a single varbind to the current varbind set. The $value provided in either call is taken to be an unsigned, 32-bit integer.

AddCounterVarbind($value)

Appends a new varbind to the current varbind set. The OID of the new varbind is automatically generated and the varbind’s type is set to “counter”.

AddCounterVarbind($oid, $value)

Appends a new varbind to the cvurrent set. The OID is set to the contents of $oid; the varbind's type is set to "counter", and the varbind's value is set to the contents of $value.

The call is the equivalent of AddVarbind($oid, "counter", $value);

AddCounter64Varbind

```
AddCounter64Varbind( $value )

        AddCounter64Varbind( 1843 );



        my $reallybignumber = "1002003004005006007 00";

        AddCounter64Varbind( $reallybignumber );



AddCounter64Varbind( $oid, $value )

        AddCounterVarbind( "1.3.6.1.4.1.78.14.30.44",  $numPollsIssued  );
```

The two AddCounter64Varbind() functions append a single varbind to the current varbind set. The $value provided in either call is taken to be an unsigned, 64-bit integer.

AddCounter64Varbind is identical to AddCounterVarbind except that the contents of $value can contain far larger values.

> Care must be taken with how 64-bit signed and unsigned values are handled by your Perl code. The Perl interpreter, upon recognizing these values as numeric will convert them to Perl's floating point data type. You will then lose the initial representation. To avoid this, have your Perl logic handle these large values as if they are character strings.

AddCounter64Varbind($value)

Appends a single varbind to the current set of varbinds. The $value is expected to be a non-negative whole number. The OID for the varbind is automatically generated by the API. The type for the varbind is set to "counter64".

AddCounter64Varbind($oid, $value)
Appends a single varbind to the current set of varbinds. The $value is expected to be a non-negative whole number. The OID for the varbind is set according to the contents of $oid. The type for the varbind is set to "counter64".

The call is the equivalent of AddVarbind($oid, "counter64", $value);

AddIPAddressVarbind

```
AddIPAddressVarbind( $value )

        AddIPAddressVarbind(  "192.168.1.1"  );


        my $ipaddress = "10.1.1.4";

        AddCounter64Varbind( $ipaddress );


AddIPAddressVarbind( $oid, $value )

        AddCounterVarbind( "1.3.6.1.4.1.78.0.11",  $ipaddress  );
```

The AddIPAddress() calls add a single varbind to the current varbind set. The contents of $value must be an IPv4 Address of the form "xxx.xxx.xxx.xxx". Do not pass a hostname as $value, use Perl's name resolution calls to convert a hostname to an IPv4 address first. Also, do not pass an IPv6 value as the contents of $value.

These two calls add the varbind as being an "ipaddr", one of SNMP's data types as seen in Table 2 and Table 3. This data type supports only IPv4 addresses. SNMP does not support IPv6 addresses using this data type.

AddIPAddressVarbind($value)
Adds one varbind, in the form of an IPv4 Address, to the current varbind set. The contents of $value are to be a character string of the form "xxx.xxx.xxx.xxx" (ex: "100.101.254.3" or "18.4.49.200"). The OID is automatically generated and the type is set to "ipaddr".

AddIPAddressVarbind($oid, $value)
Adds one varbind, in the form of an IPv4 Address, to the current varbind set. The contents of $value are to be a character string that represents a single IPv4 Address. The value of $oid is taken to the varbind's OID value. The type is set to "ipaddr".

The call is the equivalent of AddVarbind($oid, "ipaddr", $value);

AddOIDVarbind

```
AddOIDVarbind( $value )

        AddOIDVarbind(  "192.168.1.1"  );  # Yes, IPAddrs look like OIDs

        AddOIDVarbind( "1.2.3.4.5" );


        my $cisco = "1.3.6.1.4.1.9";

        AddOIDVarbind( $cisco );


AddOIDVarbind( $oid, $value )

        AddOIDVarbind( "1.3.6.1.4.1.78.0.1",  "1.3.6.1.4.1.78"  );
```

The two AddOIDAddress() calls add a single varbind to the current varbind set. The contents of $value must be an OID sequence, prepared as a character string. For example, "1.3.6.1.4.1".

For either call the contents of $value need to be a dot-delimited numeric sequence. Each element of the sequence must be valid, as discussed in the section SNMP Naming.

AddOIDVarbind($value)

Adds $value in the form of an OID varbind to the current varbind set. The OID for the varbind is automatically generated and the type is set to "oid".

AddOIDVarbind($oid, $value)

Adds $value in the form of an OID varbind to the current varbind set. The OID for the varbind is taken from $oid and the value of the varbind is taken from $value.

The call is the equivalent of AddVarbind($oid, "oid", $value);

AddTimeTicksVarbind

```
AddTimeTicksVarbind( $value )

          AddTimeTicksVarbind(  system.sysUpTime );

          AddTimeTicksVarbind( 400 );


          my $oneMinuteFromNow = system.sysUpTime + ( 100 * 60 );

          AddTimeTicksVarbind( $oneMinuteFromNow );


AddTimeTicksVarbind( $oid, $value )

          AddTimeTicksVarbind( “1.3.6.1.4.1.78.0.2”,  “12345”  );
```

AddTimeTicksVarbind() calls add a single varbind to the current varbind set. The contents of $value must be within the range of an unsigned, 32-bit integer.

The SNMP TimeTicks data type represents a span of time, measured in hundredths of a second. 100 ticks equals 1 second.

The ‘sysUpTime’ MIB variable from the ‘system’ group reports the life span of the reporting SNMP Agent using TimeTicks.

AddTimeTicksVarbind($value)

Adds a single varbind to the current varbind set. The contents of $value represents a time period, measured in 100ths of a second. The range is a non-negative whole number up to the maximum allowed for a 32-bit unsigned integer. The OID is autogenerated by the API and the type is set to “timeticks”.

AddTimeTicksVarbind($oid, $value)

Adds a single varbind to the current varbind set. The OID is set to $oid and the varbind’s value is set to $value.

The call is the equivalent of AddVarbind($oid, “timeticks”, $value);

AddUnsignedVarbind

```
AddUnsignedVarbind( $value )

        AddUnsignedVarbind( 0 );

        AddUnsignedVarbind( 1000000 );


        my $year = 2012;

        AddUnsignedVarbind( $year );


AddUnsignedVarbind( $oid, $value )

        AddUnsignedVarbind( “1.2.3.4.5”,  1000000 );
```

The AddUnsignedVarbind() calls add a single varbind to the varbind set. In each case the value assigned into the varbind is an unsigned, 32-bit integer. The range, then, is 0 up to 4,294,967,295, inclusive.

AddUnsignedValue($value)

Appends one varbind to the current varbind set. The contents of $value are taken to be an unsigned 32-bit integer. The varbind’s OID it auto-generated by the API and the type is set to “unsigned”.

AddUnsignedValue($oid, $value)

Appends one varbind to the current varbind set. The contents of $value are taken to be an unsigned 32-bit integer. The varbind’s OID is set to $oid and the type is set to “unsigned”.

The call is the equivalent of AddVarbind($oid, “unsigned”, $value);

TraceAddVarbinds

AddTraceVarbinds($setting)

LogMatrix Technical Support

LogMatrix is committed to offering the industry's best technical support to our customers and partners. You can quickly and easily obtain support for NerveCenter, our proactive IT management software.

Professional Services

LogMatrix offers professional services when customization of our software is the best solution for a customer. These services enable us, in collaboration with our partners, to focus on technology, staffing, and business processes as we address a specific need.

Educational Services

LogMatrix is committed to providing ongoing education and training in the use of our products. Through a combined set of resources, we can offer quality classroom style or tailored on-site training.

Contacting the Customer Support Center

Telephone Support

Phone: 1-800-892-3646 or 1-508-597-5300

E-mail support

E-mail: techsupport@logmatrix.com.

Electronic Support

LogMatrix has a Web-based customer call tracking system where you can enter questions, log problems, track the status of logged incidents, and check the knowledge base.

When you purchased your product and/or renewed your maintenance contract, you would have received a user name and password to access the LogMatrix Call Tracking System using SalesForce. You may need to contact your contracts or NerveCenter administrator for the username and password for your account with SalesForce.

If you have not received or have forgotten your log-in credentials, please e-mail us with a contact name and company specifics at techsupport@logmatrix.com.

We are committed to providing ongoing education and training in the use of our products. Through a combined set of resources, we offer quality training to our global customer base.

Online Access

For additional NerveCenter support information, please go the LogMatrix website www.logmatrix.com for access to the following sections of information.

- Patches and Updates – latest installation files, patches and updates including documentation for NerveCenter.
- Software Alerts – latest software alerts relative to NerveCenter.

User Community Access

You can seek as well as share advice and tips with other NerveCenter users at http://community.logmatrix.com/LogMatrix/ .