



LogMatrix®
NerveCenter™

Designing and Managing Behavior Models

Version 6.2

COPYRIGHT

Portions ©1989-2017 LogMatrix, Inc. All rights reserved.

DISCLAIMERS

LogMatrix, Inc. (“LogMatrix”) makes no representations or warranties, either expressed or implied, by or with respect to anything in this manual, and shall not be liable for any implied warranties of merchantability or fitness for a particular purpose or for any indirect, special or consequential damages.

These applications are available through separate, individual licenses. Not every feature or application described herein is licensed to every customer. Please contact LogMatrix if you have licensing questions.

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of LogMatrix. While every precaution has been taken in the preparation of this book, LogMatrix assumes no responsibility for errors or omissions. This publication and the features described herein are subject to change without notice.

The program and information contained herein are licensed only pursuant to a license agreement that contains use, reverse engineering, disclosure and other restrictions.

TRADEMARKS

LogMatrix is registered in the U.S. Patent and Trademark Office. NerveCenter and the LogMatrix Logo are trademarks of LogMatrix, Inc.

All other products or services mentioned in this manual may be covered by the trademarks, service marks, or product names as designated by the companies who market those products.

LogMatrix, Inc.
230 N. Serenata Drive, Suite 711
Ponce Vedra Beach, FL 32082 USA

Toll Free +1 (800) 892-3646
Phone +1 (508) 597-5300
Fax +1 (774) 348-4953

info@logmatrix.com
<http://www.logmatrix.com>

Introduction	1
NerveCenter Documentation	1
LogMatrix Technical Support	4
Understanding NerveCenter	5
What is NerveCenter?	5
How NerveCenter Manages Nodes	6
Defining a Set of Nodes	6
Detecting Conditions	6
Correlating Conditions	7
Responding to Conditions	12
Main NerveCenter Components	15
The NerveCenter Server	16
The NerveCenter Database	16
NerveCenter Interfaces	19
Role in Network Management Strategy	22
Standalone Operation	23
Using Multiple NerveCenter Servers	24
Integration with Network Management Platforms	24
Behavior Models and Their Components	27
Behavior Models	27
Detecting Conditions	28
Tracking Conditions	28
Monitoring a Set of Nodes	29
NerveCenter Objects	30
Nodes	30
Property Groups and Properties	36
Polls	37

Trap Masks	39
Alarms	41
Alarm Scope	42
NerveCenter and Perl	45
Constructing Behavior Models	47
How the Pieces Fit Together	48
An Example of a Behavior Model	50
Getting Started with the NerveCenter Client	53
Starting the NerveCenter Client	53
Connecting to a Server	54
Connecting to a Server Manually	55
Connecting to a Server Automatically	57
Sharing MIB Information from Multiple Servers	59
Selecting the Active Server	60
Deleting a Server from the Server List	60
Changing the Server Port on the Client	61
Setting Up Alarm-Instance Filters	62
Filtering Alarms by IP Range	63
Filtering Alarms by Severity	70
Filtering Alarms by Property Groups	74
Associating a Filter with a Server	77
Rules for Associating Filters with Alarms	79
Specifying Heartbeat Messaging	80
Modifying the Heartbeat Message Interval	81
Deactivating Heartbeat Messaging	82
Disconnecting from a Server	83
Discovering and Defining Nodes	84
Discovering Nodes	84
Using IPSweep Behavior Model	85
Defining Nodes Manually	92
IPv6 and NerveCenter	97

Configuring SNMP Settings for Nodes	99
Manually Changing the SNMP Version Used to Manage a Node	99
Changing the Security Level of an SNMPv3 Node	102
Changing the Authentication Protocol for an SNMPv3 Node	105
Classifying the SNMP Version Configured on Nodes	107
Classifying the SNMP Version for One or More Nodes Manually	109
Classifying the SNMP Version for All Nodes Manually	110
Confirming the SNMP Version for a Node	111
When NerveCenter Classifies Node SNMP Versions	114
How NerveCenter Classifies a Node SNMP Versions	115
NerveCenter Support for SNMPv3	117
Overview of NerveCenter SNMPv3 Support	117
NerveCenter Support for SNMPv3 Security	118
NerveCenter Support for SNMPv3 Digest Keys and Passwords	119
Viewing the SNMPv3 Operations Log	120
Signing a Log for SNMPv3 Errors Associated with Your Client	121
Signing a Log for SNMPv3 Errors Associated with a Remote Client or Administrator	122
Viewing the SNMPv3 Operations Log	123
SNMP Error Status	124
Using the SNMP Test Version Poll	126
Testing SNMPv1 and v2c Agents	126
Testing SNMPv3 Agents	126
How To Use the Test Version Poll	127
Defining Property Groups and Properties	129
Listing Property Groups and Properties	129
Listing Property Groups	130
Listing Properties	131
Creating a Property	132
Creating a New Property Group	133
Based on an Existing Property Group	134

Based on the Contents of MIBs	135
Adding Properties Manually	138
The nl-ping Property	138
Filtering Properties	139
Assigning a Property Group to a Node	143
Using the Node Definition Window	144
Using the Node List Window	145
Using the AssignPropertyGroup() Function	147
Using the Set Attribute Alarm Action	155
Using OID to Property Group Mappings	158
Tips for Using Property Groups and Properties	159
Categorizing Nodes	160
Move from the General to the Specific	160
MIB Objects	160
Using Polls	161
Listing Polls	162
Defining a Poll	164
Writing a Poll Condition	166
The Basic Procedure for Creating a Poll Condition	167
Functions for Use in Poll Conditions	168
Using the Pop-Up Menu for Perl	182
Examples of Poll Conditions	184
Documenting a Poll	186
How to Create Notes for a Poll	187
What to Include in Notes for a Poll	190
Enabling a Poll	191
Using Trap Masks	193
About Trap Masks	193
How NerveCenter Decodes SNMPv2c/v3 Traps	194
How NerveCenter Decodes ICMP Events	195
Listing Trap Masks	196
Defining a Trap Mask	198
Writing a Trigger Function	202

Functions for Use in Trigger Functions	203
Variables for Use in Trigger Functions	207
Examples of Trigger Functions	208
Documenting a Trap Mask	210
How to Create Notes for a Trap Mask	211
What to Include in Notes for a Trap Mask	214
Enabling a Trap Mask	216
Using Other Data Sources	219
Built-In Triggers	220
SNMP Requests	220
ICMP Requests	221
ICMP Responses	221
Built-in Trigger Firing Sequence	223
Matching Errors with Pending SNMP and Ping Requests	224
Multi-homed Nodes	224
Built-In Triggers	225
An Example Using Built-In Triggers	228
Another NerveCenter	229
Creating a Trap Mask	230
Variable Bindings for NerveCenter Informs	234
An Example Trigger Function	235
Using Alarms	237
Listing Alarms	238
Defining an Alarm	241
Alarm Scope	244
Defining States	245
Defining a State	247
Changing the Size of the State Icons	248
Deleting a State	249
Defining Transitions	249
Defining a Transition	250

Associating an Action with a Transition	251
Changing the Size of Transition Icons	252
Changing the Color of Transition Lines	253
Deleting a Transition	254
Documenting an Alarm	254
How to Create Notes for an Alarm	255
What to Include in Notes for an Alarm	257
Enabling an Alarm	259
Correlation Expressions	261
Alarm Actions	269
Action Router	269
Alarm Counter	271
Beep	274
Clear Trigger	275
Command	277
Delete Node	278
EventLog	279
Fire Trigger	280
Forward Trap	284
Inform	286
Inform Platform	289
Inform Specific Numbers	290
Log to File	291
Notes	292
Perl Subroutine	294
Defining a Perl Subroutine	295
Functions for Use in Perl Subroutines	297
NerveCenter Variables	301
Perl Subroutine Example	303
Send SMS	304
Send Trap	305
Set Attribute	308
SMTP Mail	311
SNMP Set	312

Performing Actions Conditionally (Action Router)	315
Listing Existing Action Router Rules	316
Creating an Action Router Rule	318
Defining a Rule Condition	319
Defining a Rule Action	323
Creating Multi-Alarm Behavior Models	325
IfUpDownStatusByType	325
IF-IfStatus Alarm	327
IF-SelectType Perl Subroutine	328
Interface-type Alarms	329
IF-IfFramePVC	330
IfColdWarmStart Alarm	331
IfNmDemand Alarm	332
Managing NerveCenter Objects	333
Enabling Objects	333
Copying Objects	334
Copying a Property Group	335
Copying Other Objects	335
Deleting Objects	336
Using a Delete Button	336
Using a Pop-Up Menu	337
Changing an Object Property or Property Group	338
Changing Poll or Alarm Properties	338
Changing a Node Property Group	339
Changing an Alarm Scope	339
Suppressing Polling	340
Suppressing a Node	340
Making a Poll Suppressible	341
Changing Other Node Attributes	342

NerveCenter Severities	343
Definition of a Severity	344
Severity Attributes Used by NerveCenter	345
Severity Attributes and Network Management Platforms	345
Default Severities	346
Creating a New Severity	348
Creating Custom Colors	350
Importing and Exporting NerveCenter Nodes and Objects	353
Exporting Behavior Models to Other Servers	353
Exporting Behavior Models to a File	355
More About Exporting Behavior Models	357
Exporting NerveCenterObjects and Nodes to Other Servers	357
Exporting NerveCenterObjects and Nodes to a File	360
More about Exporting Objects	362
Importing Node, Object, and Behavior Model Files	364
Importing Behavior Models or Nodes with ImportUtil	365
Communications and Data	367
Debugging a Behavior Model	371
Enabling a Behavior Model's Components	371
Checking Properties and Property Groups	371
Checking a Poll's Property	372
Checking a Poll's Poll Condition	372
Checking an Alarm's Property	373
Matching Triggers and Alarm Transitions	373
Identities of Triggers and Transitions	374
Rules for Matching	375
Examples of Matching Triggers and Transitions	376
Auditing Behavior Models	378
Behavior Model Log	379

Downstream Alarm Suppression	383
Understanding How the Model Works	383
Testing the Model	387
Importing the New Model	387
Identifying Parent-Child Relationships	388
Making the Relationship Information Available to NerveCenter	388
Testing the Alarm Suppression Model	389
Understanding the Technical Details	390
Alarms	391
Perl Subroutines	402
Error Messages	411
Alarm Filter Manager Error Messages	413
Deserialize Manager Error Messages	413
Flatfile Error Messages	413
Inform NerveCenter Error Messages	413
LogToFile Manager Error Messages	414
Poll Manager Error Messages	414
Protocol Manager Error Messages	414
PA Resync Manager Error Messages	415
Server Manager Error Messages	417
Trap Manager Error Messages	419
NerveCenter installation Error Messages	420
Index	423

Designing and Managing Behavior Models describes how NerveCenter works and how you can monitor your network most effectively. This book is written for users operating the NerveCenter Client.


NerveCenter Documentation

This section describes the available NerveCenter documentation, which explains important concepts in depth, describes how to use NerveCenter, and provides answers to specific questions.

The documentation set is provided in online (HTML) format, as well as PDF for printing or on-screen viewing.

Using the Online Help

You can view the documentation with Google Chrome, Mozilla Firefox, Apple Safari, or Microsoft Edge. Refer to the NerveCenter Release Notes for the browser versions supported with this release.

Note: For in-depth instructions on using the online documentation, click the Help button  in the upper right of the Help window.

Printing the Documentation

The NerveCenter documentation is also available as Portable Document Format (PDF) files that you can open and print. All PDF files are located in your *installpath/doc* directory.

Note: You must have Adobe Acrobat Reader to open or print the PDF files. You can download the Reader free from Adobe's Web Site at www.adobe.com.

The NerveCenter Documentation Library

The following documents ship with NerveCenter.

Book Title	Description	Application	Audience	PDF for Print
NerveCenter Release Notes	Describes new NerveCenter features and includes late-breaking information, software support, corrections, and instructions.	All	All	relnotes.pdf
Installing NerveCenter	Helps you plan and carry out your NerveCenter upgrades and new installations. Use the <i>Release Notes</i> in conjunction with this book.	All	Installation team	install.pdf
Managing NerveCenter	Explains how to customize and tune NerveCenter after it has been installed.	NerveCenter Administrator	Administrator	managing_ nervecenter.pdf
NerveCenter Platform Integration Guide	Explains how to integrate NerveCenter with network management platforms.	NerveCenter Administrator	Administrator	integratingNC.pdf
Learning to Create Behavior Models	Provides step-by-step instructions and examples for creating behavior models.	NerveCenter Client	Users with administrative privileges	learningModel.pdf
Designing and Managing Behavior Models	Explains behavior models in depth, how to create or modify models, and how to manage your models.	NerveCenter Client	Users with administrative privileges	designingModels.pdf
Monitoring Your Network	Explains how NerveCenter works and how you can most effectively monitor your network.	NerveCenter Client	Users	monitoringNet.pdf

UNIX Systems

On UNIX systems, NerveCenter man pages provide command reference and usage information that you view from the UNIX shell as with other system man pages. When you specify documentation during NerveCenter installation, the script installs nroff-tagged man pages and updates your system's MANPATH environment variable to point to the NerveCenter man page directory.

Document Conventions

This document uses the following typographical conventions:

Element	Convention	Example
Key names, button names, menu names, command names, and user entries	Bold	Press Tab Enter ovpa -pc
<ul style="list-style-type: none"> ■ A variable you substitute with a specific entry ■ Emphasis ■ Heading or Publication Title 	<i>Italic</i>	Enter <i>./installdb -f IDBfile</i>
Code samples, code to enter, or application output	Code	<code>iifInOctets > 0</code>
Messages in application dialog boxes	<i>Message</i>	<i>Are you sure you want to delete?</i>
An arrow (>) indicates a menu selection	>	Choose Start > Programs > NerveCenter

Caution: A caution warns you if a procedure or description could lead to unexpected results, even data loss, or damage to your system. If you see a caution, proceed carefully.

Note: A note provides additional information that might help you avoid problems, offers advice, and provides general information related to the current topic.

Documentation Feedback

LogMatrix, Inc. is committed to providing quality documentation and to helping you use our products to the best advantage. If you have any comments or suggestions, please send your documentation feedback to:

Documentation
 LogMatrix, Inc.
 230 N. Serenata Drive, Suite 711
 Ponce Vedra Beach, FL 32082 USA
support@logmatrix.com

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

For Telephone Support

Phone: Toll Free +1 (800) 892-3646 or Phone +1 (508) 597-5300

For E-mail Support

E-mail: support@logmatrix.com.

This chapter explains:

- What type of product NerveCenter™ is
- How NerveCenter manages nodes
- What the NerveCenter main components are
- What roles NerveCenter can play in a network or system management solution

What is NerveCenter?

There are many network management tools designed to identify network faults and send alerts, but in doing so they may often flood the event console with raw data. Each critical or warning message indicating a potential problem — a failed router, for example — is usually accompanied by many additional messages regarding devices downstream that cannot be contacted. As a result, there is a great deal of information to sift through before the real problem can be identified and corrected.

NerveCenter is a network management tool that helps automate the identification, tracking, management, and resolution of important network events to facilitate proactive isolation and response to key events. NerveCenter uses the Simple Network Management Protocol (SNMP) to acquire data about managed devices, as well as Internet Control Message Protocol (ICMP) messages from your network to provide basic information about unresponsive devices.

For each device being monitored, NerveCenter's event correlation engine creates one or more finite state machines — or *alarms* — that define operational states of interest and the transitions between those states. These state machines enable NerveCenter to correlate data from multiple sources over time before concluding that a problem exists. As a simple example, an interface link-down trap might not indicate a problem if a link-up trap is received within a given amount of time, allowing both traps to be ignored. If the link-up trap does not follow in the given time, then NerveCenter can then implement predefined actions such as notifying an administrator, executing a script to correct the problem, or notifying a network management platform.

In addition to being an advanced event automation solution, NerveCenter is also a highly scalable cross-platform client/server application. It can run co-resident with a network management platform (such as IBM Tivoli Netcool/OMNIBus) and manage thousands of nodes, or distributed as a background process at tens or even hundreds of remote offices.

How NerveCenter Manages Nodes

To perform its job of event automation, NerveCenter relies on the definition of *behavior models*. These models are constructed from NerveCenter objects (which we'll discuss in detail later) and define:

- Which nodes the behavior model will affect
- How NerveCenter will detect certain conditions on these nodes
- How NerveCenter will correlate the conditions it detects
- How NerveCenter will respond to network problems

The following sections elaborate on the tasks that NerveCenter performs in order to automate event handling.

Defining a Set of Nodes

NerveCenter can get the list of devices to monitor from a network management platform, discover them on the network, or import this information from another NerveCenter database.

NerveCenter assigns to each managed node a set of *properties*, and these properties determine which behavior models apply to a node. Properties typically describe the type of the device—for example, a router—or are named after objects in the management information base (MIB) used to manage the node.

Once NerveCenter assigns a set of properties to a node, NerveCenter automatically applies to that node all of the models that refer to those properties. If NerveCenter detects that a node has been deleted or that its properties have changed, the product immediately retires or updates the set of models that are actively managing that node. This dynamic process enables NerveCenter to adapt at once to changes in network configuration reported by the management platform or by NerveCenter's own discovery mechanism.

It is also possible to assign properties to nodes manually to further refine the set of models that NerveCenter uses to manage a node. For example, you may want to distinguish a backbone router from a campus router to regulate how much and how often status information is collected.

Detecting Conditions

As is discussed in the section [Role in Network Management Strategy on page 22](#), NerveCenter can collect network and system data from a variety of sources. However, most frequently NerveCenter obtains data from Simple Network Management Protocol (SNMP) agents running on managed nodes. This means that NerveCenter detects most conditions by:

- Receiving and interpreting an SNMP trap
- Polling an SNMP agent for data and analyzing that data

One of the criticisms of SNMP-based enterprise management platforms over the years has been that, because SNMP trap delivery is unreliable, the platform must poll agents and this polling generates too much network traffic. NerveCenter helps alleviate this problem by enabling you to determine the interval at which a poll is sent and to turn a poll off. Even more important is NerveCenter's *smart polling* feature. NerveCenter sends a poll to a node only if the poll:

- Is part of a behavior model designed to manage that node
- Can cause a change in the alarm's state.

Also, because of NerveCenter's client/server architecture, NerveCenter servers can be distributed so that all polling is done on LANs, and not across a WAN. Furthermore, use of SNMP v2c and v3 features allow SNMP to be utilized both reliably and securely.

Correlating Conditions

Event correlation involves taking a number of detected network conditions, often a large number, and determining:

- How these conditions, or some subset of them, are related
- The underlying cause of a set of conditions, or the problem to which they have led

For instance, NerveCenter may look at a large number of events and identify a subset of events that relate to SNMP authentication failures on a managed node. NerveCenter may then determine that the authentication failures were far enough apart that no problem exists, or it may find that several failures occurred within a short period of time, indicating a possible security problem. In the latter case, NerveCenter might notify administrators of the potential problem. In this way, administrators receive one notice about a potential security problem rather than having to browse through a long list of detected conditions and identify the problem themselves.

Detected conditions can be correlated in many ways. In fact, once you start working with NerveCenter, you will help determine how these conditions are correlated yourself. However, there are some typical ways in which NerveCenter finds relationships between conditions. Several of these methods are discussed in the following sections:

- [Detecting the Persistence of a Condition on the next page](#)
- [Finding a Set of Conditions on page 9](#)
- [Looking for a Sequence of Conditions on page 10](#)

Detecting the Persistence of a Condition

Probably the simplest method of correlating detected conditions is to search for the persistence of a problem. For example, a network administrator might want to know if an SNMP agent sends a link-down trap and that trap is not followed within three minutes by a link-up trap. NerveCenter can track such a link-down condition using a state diagram similar to the one shown below.

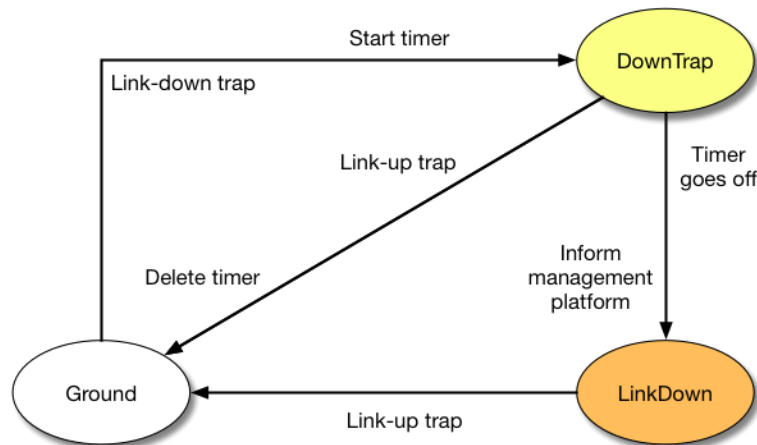


Figure 1: State Diagram for Detecting a Link-Down Condition

Let's say that NerveCenter has this state diagram in memory and is tracking a particular interface for a link-down condition.

- The first time NerveCenter sees a link-down trap concerning that interface, the current state becomes DownTrap, and NerveCenter starts a three-minute timer.
- If NerveCenter receives a link-up trap within three minutes of the link-down trap, the current state reverts to Ground (normal) because NerveCenter is looking for a *persistent* link-down condition. In addition, NerveCenter stops the timer. However, if three minutes expire before a link-up trap arrives, the current state becomes LinkDown, and NerveCenter informs a network management platform that the link is down.
- The current state remains LinkDown until a link-up trap does arrive. At that point, the current state reverts to Ground, and the process begins again.

Finding a Set of Conditions

Another common type of event correlation is the identification of a set of conditions. For example, let's say that you're monitoring the interfaces on a router. To be notified when a low-speed interface goes down or when a high-speed interface goes down, you might use the following state diagram.

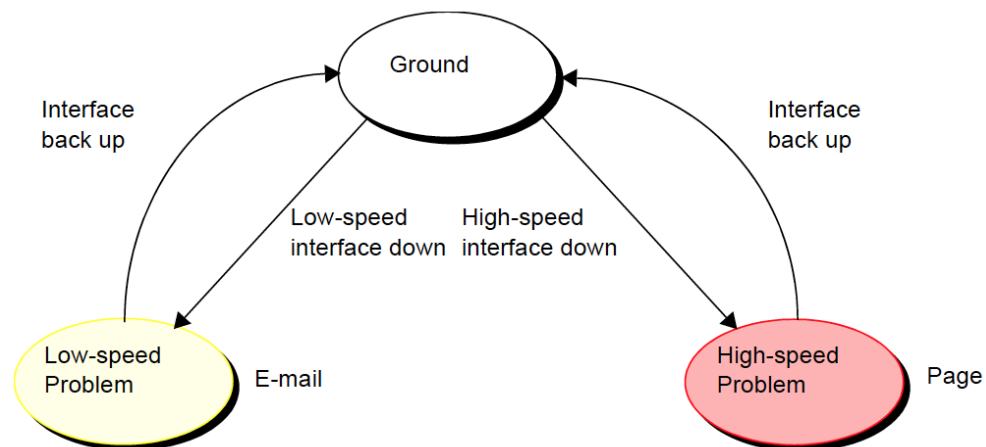


Figure 2: State Diagram for Detecting a Router Interface Problem

What causes state transitions in this situation? NerveCenter can poll the SNMP agent on the router for the values of the following interface attributes: `ifOperStatus`, `ifAdminStatus`, `ifSpeed`, `ifInOctets`, and `ifOutOctets`.

If the poll successfully returns values for these attributes, NerveCenter can then evaluate the expression shown below in pseudocode:

```

if ifOperStatus is down && ifAdminStatus is up &&
  (ifInOctets > 0 || ifOutOctets > 0)
  if ifSpeed < 56K
    move to lowSpeedProblem state
  else
    move to highSpeedProblem state
else
  move to ground state
  
```

This code is looking for two sets of conditions. The first set is:

- The operational state of the interface is down.
- The administrative status of the interface is up.
- Traffic has been passed on this interface. (If no traffic has been passed, the interface is just coming up.)
- The interface's current bandwidth is less than 56K.

If this set of conditions is met, a problem exists on an interface that is probably used for a dial-up connection.

The second set of conditions is the same as the first, except that the last condition is that the interface's current bandwidth is greater than or equal to 56K. If this set of conditions is met, a problem exists on a higher speed interface.

If neither of these sets of conditions is met, the current state should return to, or remain at, Ground.

NerveCenter may detect many conditions concerning an interface before it finds the set of conditions it is looking for. The administrator need not see information about each of these conditions. He or she will be emailed or paged if the interface goes down.

Looking for a Sequence of Conditions

NerveCenter also enables you to correlate conditions by looking for sequences of conditions. This type of correlation is possible because, in NerveCenter, each state in a state diagram can look for a different set of conditions. For instance, let's look at a state diagram that NerveCenter uses to track the status of a node and its SNMP agent. The diagram includes states for the following conditions:

- The node and its SNMP agent are up.
- The node is up, but its agent is down.
- The node is unreachable.
- The node is down.

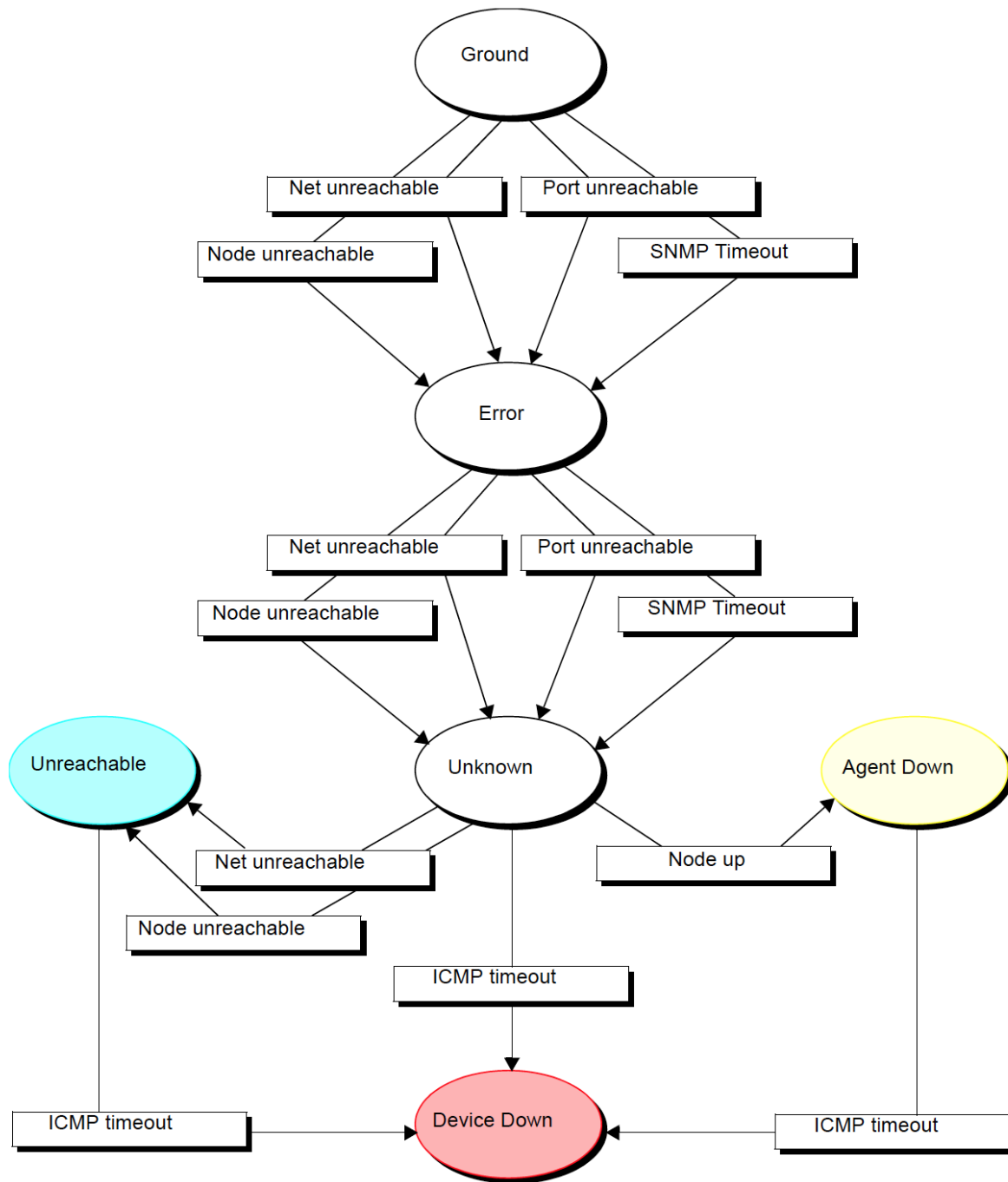


Figure 3: State Diagram for Determining Node Status

Note: A more realistic state diagram for tracking the status of a node would include transitions from the terminal problem states back to Ground.

When checking the status of a node and its SNMP agent, NerveCenter begins by polling the node to see if the node's SNMP agent will return the value of the MIB attribute `sysObjectID`. If the agent returns this value, the current state remains Ground. However, NerveCenter makes Error the current state if:

- The node, or the network the node is on, is unreachable
- The node is reachable, but the SNMP agent doesn't respond

Similarly, NerveCenter changes the current state to Unknown if it detects for a second time that the node is unreachable or the node's SNMP agent isn't responding.

Once the current state becomes Unknown, though, NerveCenter begins looking for a different set of conditions. NerveCenter checks to see whether the node will respond to an ICMP ping. If it will, NerveCenter knows that the node is up, but its SNMP agent is down. If it receives another network- or node-unreachable message, NerveCenter knows that the node is unreachable. And if the ping times out, NerveCenter knows that the node is down.

This ability of different states to monitor different conditions gives you the ability to correlate *sequences* of conditions. That is, a sequence of two SNMP timeouts followed by a Node up indicates that the node is up but its agent is down. And a sequence of two Node unreachables followed by an ICMP timeout indicates that the node is down.

Responding to Conditions

NerveCenter not only enables you to detect network and system problems, but is able to respond automatically to the conditions it detects. To set up these automated responses, you associate *actions* with state transitions.

The possible actions you can define are discussed in the following sections:

- [Notification on the facing page](#)
- [Logging on the facing page](#)
- [Causing State Transitions on page 14](#)
- [Corrective Actions on page 14](#)
- [Action Router on page 15](#)

Notification

If a particular network or system condition requires the attention of an administrator, the best action to take in response to that condition is to notify the appropriate person. NerveCenter lets you notify administrators of events in the following ways:

- You can send an audible alarm (a beep) to workstations running the NerveCenter Client.
- You can send a message to an administrator using either SMS service or SMTP mail.
- You can page an administrator.
- You can send information about a network or system condition to another NerveCenter server. This capability is useful if you have a number of NerveCenter servers at different sites and want these servers to forward information about important events to a central server.
- You can send information about a network or system condition to a network management platform such as IBM Tivoli's Netcool/OMNIbus. Administrators can then be notified of a problem found by NerveCenter using the other management tool's console.

For more information on integrating NerveCenter with other network management products, see the section [Role in Network Management Strategy on page 22](#).

Logging

If you want to keep a record of an event that takes place on your network, you must explicitly log information about the event at the time it occurs. NerveCenter provides three actions that provide for such logging:

- Log to File
- system log (syslog)

Log to File writes information about an event to a file. The EventLog action writes information about an event to the system log.

When you assign a logging action to a behavior model, you have the choice of logging default data or customizing what data you deem relevant. This saves disk space and streamlines information used later for analysis and reporting.

Causing State Transitions

In some behavior models, one alarm needs to cause a transition in another. The action that enables such communication between alarms is called Fire Trigger. This action creates a NerveCenter object called a trigger that can cause a state transition in the alarm from which it was fired or in another alarm.

The Fire Trigger action also lets you specify a delay, so you can request that a trigger be fired in one minute or five hours. This feature is especially useful when you're looking for the persistence of a condition. Let's say that you want to look for three intervals of high traffic on an interface within a two-minute period. When your poll detects the first instance of high traffic, and your alarm moves out of the Ground state, you can fire a trigger with a two-minute delay that will return your alarm to the Ground state—unless a second and third instance of high traffic are detected.

If a third instance of high traffic is detected, you should cancel the trigger you fired on a delayed basis. You do this by adding the Clear Trigger action to the transition from the second high-traffic state to the third.

NerveCenter also includes a Send Trap action. You define the trap to be sent, including the variable bindings, and associate the action with a state transition. When the transition occurs, the trap is sent. The trap can be caught by a NerveCenter trap mask—in which case you can use Send Trap somewhat like Fire Trigger, to generate a trigger—or by any application that processes SNMP traps.

Corrective Actions

There are a number of NerveCenter actions that you can use to take corrective actions when a particular state transition occurs. These are:

- Command
- Perl Subroutine
- Set Attribute
- Delete Node
- SNMP Set

The Command action enables you to run any script or executable when a particular transition occurs.

The Perl Subroutine action enables you to execute a Perl script as a state-transition action. You first define a collection of Perl scripts and store them in the NerveCenter database; then, you choose one of your stored scripts for execution during a state transition.

The Set Attribute action enables you to set selected attributes of the NerveCenter objects used to build behavior models.

The Delete Node action deletes the node associated with the current state machine from the NerveCenter database. This action is useful if you use a behavior model to determine which nodes you want to monitor and manage.

The SNMP Set alarm action changes the value of a MIB attribute when an alarm transition occurs.

Action Router

The Action Router enables you to specify actions that should be performed when a state transition occurs *and other conditions are met*. To set up these conditional actions, you add the Action Router action to your state transition. Then, you use the Action Router tool to define rules and their associated actions.

For example, let's assume that you want to be notified about a state transition only if the transition puts the alarm in a critical state. You can define the following rule:

```
$DestStateSev eq 'Critical'
```

Then define the action you want taken if the severity of the destination state is Critical, for example, a page. You will be paged if:

- The Action Router action is associated with the current state transition
- The destination state for the transition is Critical

Action Router rules can be constructed using many variables that NerveCenter maintains; for instance, you can also construct rules based on:

- The name of the alarm
- The day of the week
- The time of day
- The name or IP address or group property of the node being monitored
- The name of the trigger that caused the state transition
- The name of the alarm's property
- The name or severity of the origin state
- The contents of a trap
- The contents of the varbind data associated with a trap or a poll

Main NerveCenter Components

NerveCenter is a distributed client/server application and includes the following components:

- [The NerveCenter Server on the next page](#)
- [The NerveCenter Database on the next page](#)
- [NerveCenter Interfaces on page 19](#)

The NerveCenter Server

The NerveCenter Server is responsible for carrying out all of the major tasks that NerveCenter performs. For example, it handles the polling of SNMP agents, creates NerveCenter objects such as the finite alarms mentioned earlier, and makes sure that state transitions occur at the appropriate times. The server also performs all actions associated with state transitions.

The server can run as a daemon on UNIX systems. This capability to run in the background has important implications with regard to using NerveCenter at remote sites. You can install the server and database at a remote office and have that server manage the local network, yet control the server (via the NerveCenter Client) from a central location. Servers located at remote sites can forward noteworthy information to a server at the central location as required.

The NerveCenter Database

The NerveCenter database is primarily a repository for the NerveCenter objects that make up a set of behavior models. The principal objects used in these models are:

- Nodes
- Property groups and properties
- Polls
- Trap masks
- Alarms

For brief explanations of what these objects are and how they are used, see [Objects in the Database below](#).

A set of objects that define many useful behavior models ships with NerveCenter and is available as soon as you've installed the product. For a list of these predefined behavior models, see the section [Predefined Behavior Models on page 18](#).

On UNIX systems, the NerveCenter database is implemented as a flat file.

Objects in the Database

This section contains brief definitions of the basic objects used in the construction of behavior models.

- **Nodes** - A node represents either a workstation or a network device, such as a router. Each node has an attribute called its property group that controls which behavior models NerveCenter will employ in managing the node.

Note: Strictly speaking, a node is not part of a behavior model; rather, it is the entity managed by a behavior model.

- **Property groups and properties** - As mentioned above, each node has a property group. This property group is simply a container for a set of properties, which are strings that typically either describe the type of node or name an object in the MIB used to manage the node. It is actually a node's properties, rather than its property group, that determine whether a particular behavior model will be used to manage that node.

- **Polls** - A poll defines what MIB variables NerveCenter should request the values of, how those values should be evaluated, and what action the poll should take. If the poll takes an action, it will be to fire a *trigger*, which may cause a state transition in one of NerveCenter's finite state machines.
- **Trap masks** - A trap mask describes an SNMP trap and contains the name of a trigger. If NerveCenter receives an SNMP trap that matches the description given in the trap mask, NerveCenter fires a trigger with the name defined in the trap mask. If NerveCenter receives a trap that does not match a trap mask, it discards that trap.
- **Alarms** - NerveCenter's finite state machines are called *alarms*. Each alarm defines a set of operational states (such as Normal and Down) and transitions between the states. Transitions are effected by the receipt of the proper trigger and can have actions associated with them. If actions are associated with a transition, the server performs these actions each time the transition takes place.

Behavior Models

Once a set of managed nodes has been defined, NerveCenter's monitoring activities are controlled by a set of *behavior models*. A behavior model is the group of NerveCenter objects required to detect and take action upon a single network condition, such as high traffic on an interface.

The central object in each behavior model is a deterministic finite state machine called an *alarm*. For instance, the alarm shown in Figure 4 tracks the level of traffic on an interface.

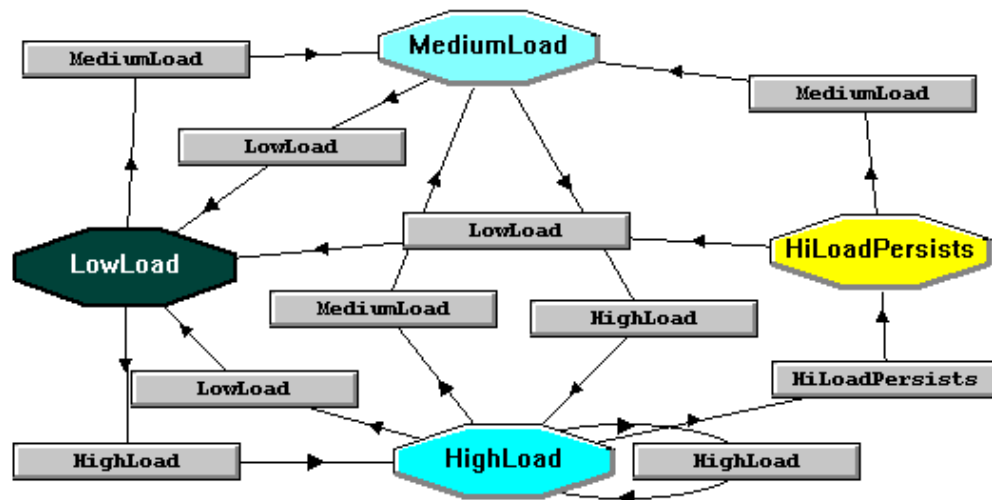


Figure 4: Alarm State Diagram

The possible states in this alarm are low, medium, and high. And these states have the *severities* Normal, Medium, and High, respectively. (The color of each state denotes its severity.) The gray rectangles in the alarm represent *state transitions*.

What about the inputs and outputs of the state machine? The inputs are called *triggers* and can come from several sources. For example, one predefined NerveCenter poll queries the SNMP agent on a device for the level of traffic on, and the capacity of, each interface on the device. If the level of use exceeds a certain percentage of the capacity for an interface, the poll fires the trigger `mediumLoad`, which can cause a state transition in an alarm.

The outputs of an alarm are called *alarm actions*. These actions are associated with the transition from one state to another by the designer of a behavior model, and NerveCenter performs these actions each time the transition occurs. There are many possible actions, including the following:

- Sending an audible alert to the workstation on which the NerveCenter Client is being run
- Executing a program or script
- Deleting a node from the NerveCenter database
- Informing a network management platform of a condition
- Logging information to a disk file
- Sending mail to an administrator
- Paging an administrator
- Sending an SNMP trap
- Setting a MIB attribute

Predefined Behavior Models

When you install NerveCenter and create a new database, that database contains the objects that make up a number of predefined behavior models. These include behavior models for:

- Detecting authentication failures
- Monitoring the error rate on network interfaces
- Monitoring link-up and link-down traps
- Monitoring the amount of traffic on network interfaces
- Indicating the status of network interfaces: up, down, and so on
- Detecting errors that inhibit accurate SNMP device management
- Determining whether a device is down, unreachable, or up with/without an agent
- Giving early warning concerning TCP connection saturation
- Verifying that the current TCP retransmission algorithm is the most efficient
- Categorizing devices based on TCP retransmission activity
- Logging information about SNMP traps

NerveCenter also includes predefined behavior models that you can import to monitor specific vendors' devices and additional models for troubleshooting, interface status, data collection, and downstream alarm suppression. For more information about behavior models, see "[Behavior Models and Their Components](#)" on page 27.

NerveCenter Interfaces

This section introduces NerveCenter's principal user interfaces:

- "The NerveCenter Administrator" below
- The NerveCenter Client Console
- "The NerveCenter Client" on page 21
- "The Command Line Interface" on page 22

The NerveCenter Administrator

Users with administrator privileges can use the NerveCenter Administrator interface to:

- Configure NerveCenter discovery mechanisms
- Configure the number of SNMP polling retries and the retry interval
- Configure NerveCenter mail and paging actions
- Manage NerveCenter log files
- Configure NerveCenter to work with a network management platform

Figure 5 shows the NerveCenter Administrator.

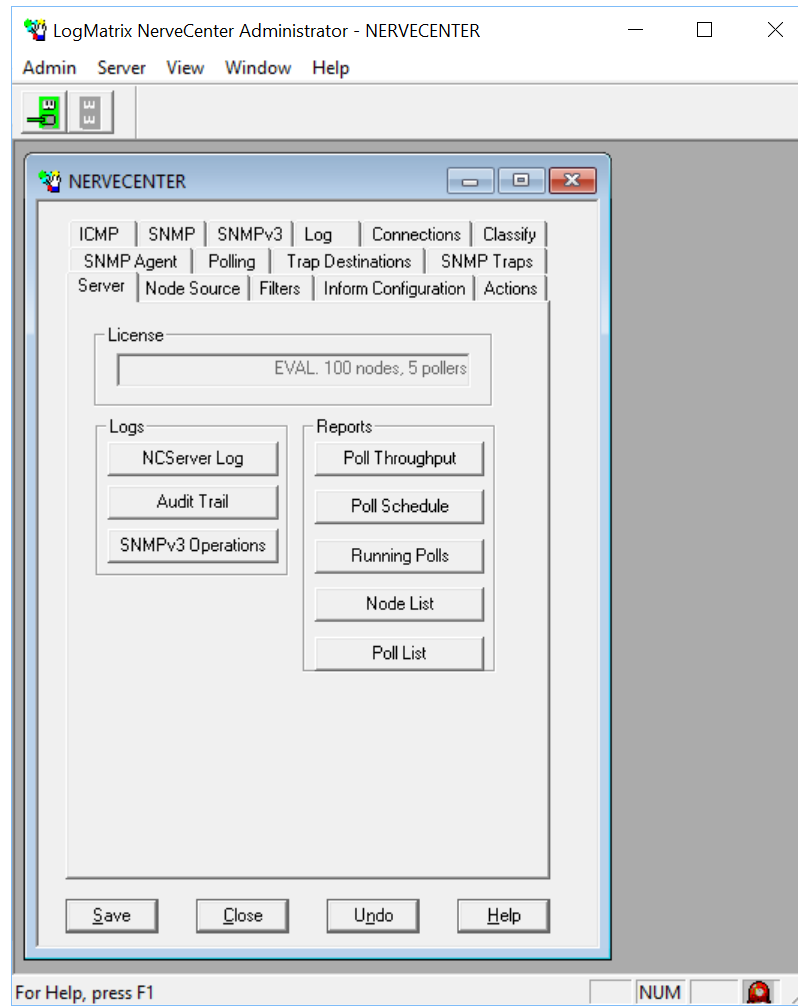


Figure 5: NerveCenter Administrator

The NerveCenter Client

The figure below shows the GUI for the NerveCenter Client.

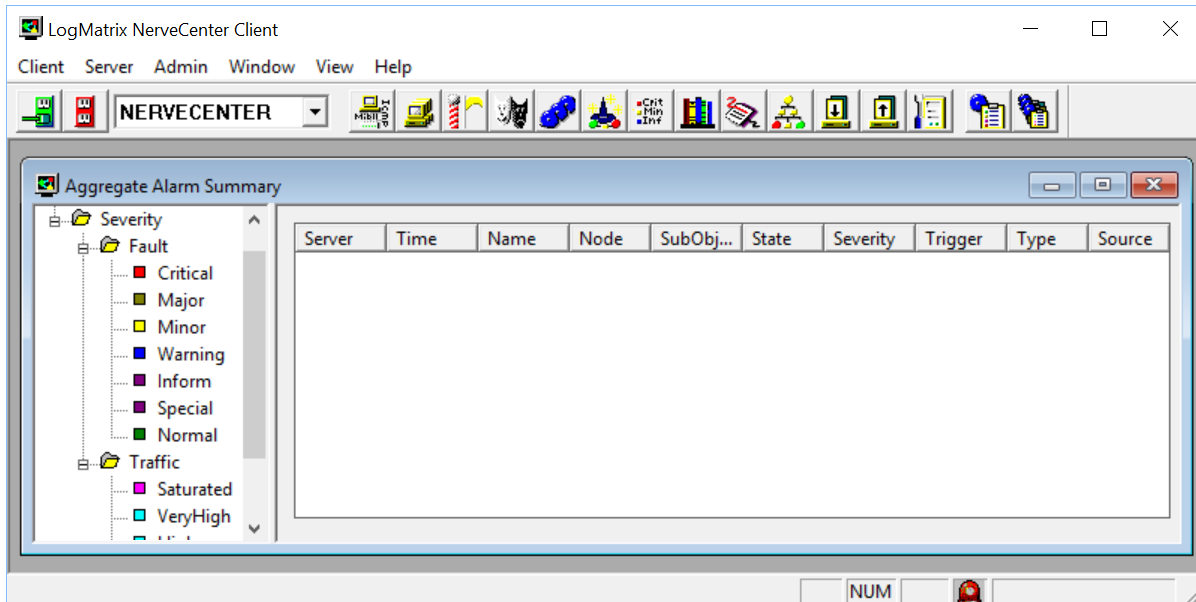


Figure 6: NerveCenter Client

Two types of users run the NerveCenter Client. Users with NerveCenter User privileges can run the client to:

- Monitor active alarms
- Filter alarms for the alarm summary windows
- View an alarm's history
- Reset alarms
- Monitor the state of managed nodes
- Generate reports

For complete information on using the NerveCenter Client to perform the tasks listed above and others, see *Monitoring Your Network*.

Users with NerveCenter Administrator privileges can perform all the tasks that users with User privileges can. In addition, they can use the client to:

- Create new behavior models
- Customize the predefined behavior models
- Modify, copy, or delete any object in the NerveCenter database

The Command Line Interface

You can use NerveCenter's command line interface (CLI) to delete, list, or set (enable or disable) alarms, trap masks, nodes, and polls from a Windows Command Prompt or a UNIX shell. You can also connect to, display the status of, and disconnect from NerveCenter servers using the CLI. You can issue commands manually or from a script.

See the appendix [Controlling NerveCenter from the Command Line](#) for command descriptions.

Role in Network Management Strategy

NerveCenter can play a variety of roles in an overall network management strategy. The role that NerveCenter plays in your strategy depends largely on the size of your network and on what other products you are using to manage your network and systems:

- If you are managing a small network, NerveCenter can be used as a standalone system. It can discover the workstations and network devices on the network, detect and correlate network conditions, respond automatically to conditions, and display information about active alarms. See the section "[Standalone Operation](#)" on the facing page for further information.
- For larger networks, multiple NerveCenters can be used in concert. Local NerveCenter systems could be set up to manage remote sites, and the local NerveCenter servers could forward important information to the NerveCenter server at the central site. See the section "[Using Multiple NerveCenter Servers](#)" on page 24 for further information.
- NerveCenter can be used in conjunction with a network management platform such as IBM Tivoli Netcool/OMNIbus which manages systems systems, networks, intranets, and databases. NerveCenter can be configured to receive messages from or send messages to this and similar network management platforms. See the section "[Integration with Network Management Platforms](#)" on page 24 for further information.

Standalone Operation

At smaller sites, you can use NerveCenter alone for your network management tasks. As we've seen, NerveCenter is very strong in the areas of event correlation and automated actions. In addition, NerveCenter includes an alarm console, as shown in [Figure 7](#).

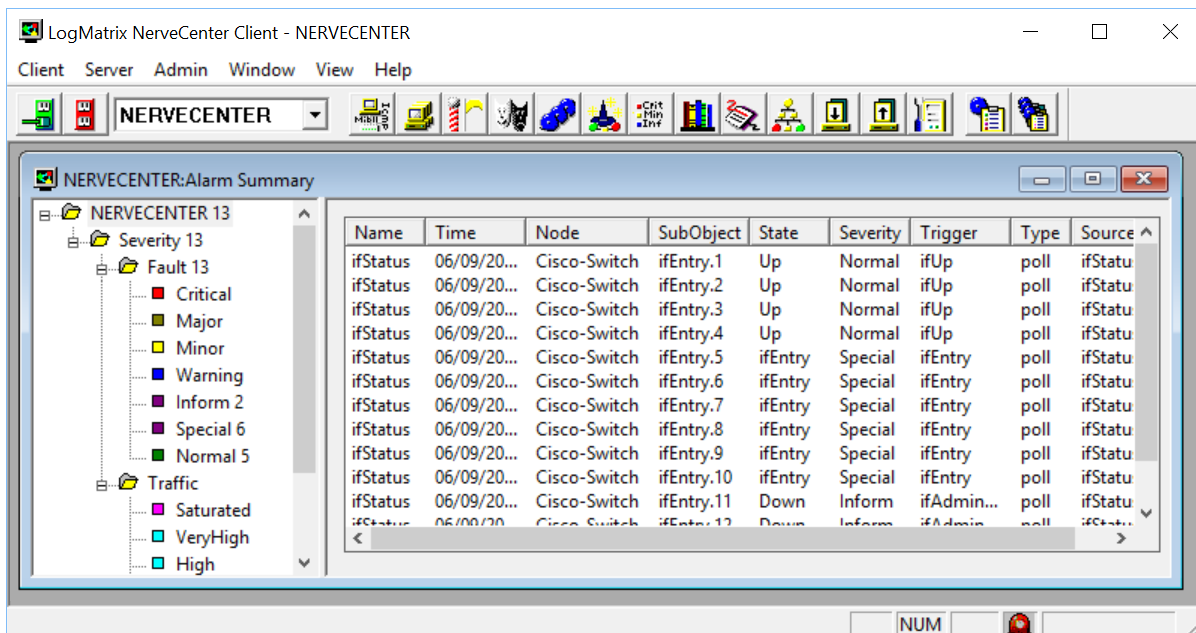


Figure 7: NerveCenter's Alarm Console

This console displays information about every current alarm instance. In addition, if you double-click on a line in the event console, you are taken to an Alarm History window that displays information about all of the alarm transitions that have occurred for the alarm instance you selected.

At small installations, no discovery mechanism is necessary; you can add nodes to NerveCenter manually. At somewhat larger sites, however, such a mechanism is helpful, and NerveCenter provides one in its Discovery behavior model.

Using Multiple NerveCenter Servers

Because a NerveCenter server can inform another NerveCenter server or management platform of a network condition, it's possible to set up NerveCenter servers at remote sites that notify a centrally located NerveCenter server or management platform (Figure 8).

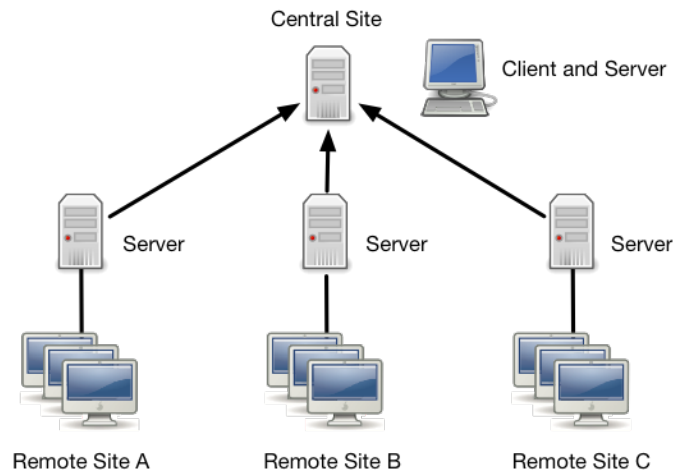


Figure 8: Distributed NerveCenter Servers

This is a reliable solution because the remote NerveCenter servers use TCP/IP to notify the central NerveCenter server and retransmit messages as necessary to ensure their delivery. There are a few advantages to this type of setup:

- Only a small amount of data is transmitted over the WAN. Any bandwidth intensive monitoring is conducted on a LAN and is managed by a remote NerveCenter server.
- Remote NerveCenter servers can be run in lights-out mode, which means that:
 - NerveCenter runs as a UNIX daemon
 - You can monitor and configure NerveCenter from a remote location
 - You can modify all NerveCenter parameters without shutting NerveCenter down
 - No display or operators are required at a site
- The central NerveCenter can further correlate and filter conditions across remote NerveCenter Server domains

Integration with Network Management Platforms

A network management platform (NMP) is an operations and problem-management solution for use in a distributed multi-vendor environment. Intelligent distributed agents on managed nodes monitor system and application log files and SNMP data. The agents apply filters and thresholds to monitored data and forward messages about conditions of interest to a central management station. When the management station receives messages, it can automatically take corrective action—such as broadcasting a command to a set of systems—or an operator can initiate a response.

You can integrate your NerveCenter installation with the NMP so that the NMP can send messages to NerveCenter for correlation or processing. After the messages arrive, NerveCenter correlates the conditions described in these messages with related conditions—from the NMP or from other sources—and can respond with any of its alarm actions, as appropriate. In addition, NerveCenter can send a message to an NMP in response to any network condition, whether the condition was originally detected by the NMP or not.

NMPs alone can detect a condition and invoke an action in response. However, you must integrate the NMP with NerveCenter if you want to:

- Correlate conditions detected by the NMP on different devices
- Correlate different types of conditions detected by the NMP on the same device
- Correlate conditions detected by the NMP with other types of events or conditions on the same device or across different devices

See the NerveCenter Platform Integration Guide for more information.

Behavior Models and Their Components

"Understanding NerveCenter" on page 5 introduced behavior models and the objects from which they're built. This chapter explains how to approach the design of a behavior model, provides detailed definitions of the NerveCenter objects used in building behavior models, and illustrates how these objects interact.

Behavior Models

For NerveCenter to detect a network condition or correlate network conditions, someone must specify *how* NerveCenter is to detect and react to one or more conditions. Such a specification is called a *behavior model*. Some behavior models ship with NerveCenter—these are called *predefined behavior models*—and others you must write to handle site-specific conditions.

When writing a behavior model, you must answer the following questions:

- What condition or conditions do I want to detect?

Although NerveCenter can receive status information from a number of sources, the most common source of such information is an SNMP agent on a managed node. Therefore, in most cases, you must decide whether the behavior model will be poll driven or event driven. That is, will you poll the agent's MIB for status information, look for SNMP traps, or both?

NerveCenter provides two objects—*polls* and *trap masks*—that enable you to get information from SNMP agents. For an overview of these objects, see the section [Detecting Conditions on the next page](#).

- What network conditions, or states, do I want to keep track of?

Each behavior model includes at least one *alarm*, and the definition of each alarm consists primarily of a state diagram. For example, an alarm that tracks the status of a managed node's SNMP agent might have the following terminal states:

- Normal
- Device Unreachable
- Agent Down
- Device Down

The state of such an alarm changes as related polls and trap masks gather new information.

For an overview of alarms, see the section [Tracking Conditions on the next page](#).

- What set of nodes do I want to manage?

A particular behavior model may not be intended for all managed devices. NerveCenter enables you to specify the set of devices that a model will manage using the following objects: *nodes*, *property groups*, and *properties*.

For an overview of the roles these objects play in a behavior model, see the section [Monitoring a Set of Nodes on the facing page](#).

Detecting Conditions

In the typical situation where your behavior model is either polling, or looking for a trap from, an SNMP agent, you detect network conditions by creating polls and trap masks.

A poll contains a poll condition that refers to a single MIB base object. For example, the following poll condition looks at an attribute of the ip base object (1.3.6.1.2.1.4):

```
if (ip.ipForwarding == 1) {
    FireTrigger("gatewayFound");
}
```

When NerveCenter polls an agent on a device, NerveCenter evaluates the poll condition against information stored in the agent's MIB. In the case of the poll condition shown above, NerveCenter would check the value of the ipForwarding attribute and compare it to 1. If the value of ipForwarding is 1—indicating that the device is a gateway—the poll generates a *trigger*. In this case, the trigger is *gatewayFound*. Every poll must be capable of generating at least one trigger.

A trap mask describes the contents of an SNMP trap. This description can be very general, such as “generic trap 4.” Or it can be very specific and include an enterprise OID, a specific trap number, and the contents of the trap's variable bindings. In either case, if the NerveCenter server receives an SNMP trap that matches the description given in a trap mask, that trap mask generates a trigger. Like the triggers generated by polls, this trigger can affect the state of one or more alarms.

Tracking Conditions

NerveCenter tracks each detected network condition using one or more alarms. The scope of an alarm is variable: an alarm can represent the state of an interface on a device, the device itself, or an entire enterprise. Many instances of an alarm can exist simultaneously.

Each alarm is basically a finite state machine. It consists of a series of states and transitions between the states. Each transition is initiated by one or more input events and can produce one or more output events. This state machine is represented in NerveCenter by a state transition diagram.

Nodes represent workstations and network devices and contain property groups, which in turn contain strings called properties. Polls and alarms are assigned properties. Given this situation, NerveCenter can enforce the following rules:

- *A poll can be sent to a particular node only if the node's property group contains the poll's property.*
- *An alarm can be instantiated for a node only if the node's property group contains the alarm's property.*

For more detailed information about the NerveCenter objects used to construct behavior models, see [NerveCenter Objects](#) below.

NerveCenter Objects

The upcoming sections provide details about the data structures of the NerveCenter objects used in the construction of behavior models. These sections not only list each object's data members, but explain how each member affects the way a behavior model functions (where appropriate). It's important to understand these details before you attempt to design a behavior model and create these objects.

The object types are discussed in the following sections:

- [Nodes](#) below
- [Property Groups and Properties](#) on page 36
- [Polls](#) on page 37
- [Trap Masks](#) on page 39
- [Alarms](#) on page 41

Nodes

In NerveCenter terminology, a node is either a workstation or a network device such as a router. NerveCenter monitors and manages a set of nodes, and each behavior model manages a subset of those nodes.

A node object has the data set shown in [Table 1](#). The table explains what information these data members contain and, where appropriate, how NerveCenter uses that information.

Note: The names of the data members shown in [Table 1](#) match the labels used in the Node Definition window, where you create and modify node objects.

SNMP Properties

Table 1: Definitions of Node Attributes: SNMP Properties

Attribute	Description
SNMP Version	The list at the upper-left corner of the SNMP tab configures how the agent is to be polled. Choose from SNMPv1, SNMPv2c, or SNMPv3 to set the level of SNMP to be used. Or select 'Unknown' to disable SNMP polling.
Port	Contains the number of the port that the node's agent uses to receive SNMP messages. By default, the port is set to 161.

Attribute	Description
Use GetBulk	Selecting this check box permits NerveCenter to use GetBulk when retrieving tables from the agent on this node when polling is done using SNMP v2c or v3. When not selected (or when polling is doing using SNMP v1), GetNext is used.
Polling	
Use Defaults	When selected, NerveCenter uses the system-wide defaults when polling the agent on this node. Clear the check box to apply node-specific parameters.
Retry Interval	The delay, in seconds, that NerveCenter will wait for a response to any issued poll request (Get, GetNext, GetBulk) before issuing another request. If Use Default is unchecked, you may enter a value from 1 to 600 (10 minutes).
Attempts	The number of poll requests NerveCenter will issue before declaring a poll to be timed out. If Use Default is unchecked, you may enter a value from 1 to 11 (an initial attempt followed by up to 10 retries).
Timeout	Displays the maximum poll timeout duration (Retry Interval x Attempts).
SNMP v1/v2c	
Read Community	Contains the community name NerveCenter will include for SNMP v1 and v2c poll requests (Get, GetNext, GetBulk) it sends to the SNMP Agent on this node. By default this value is "public". This value is not used when polling is done using SNMP v3.
Write Community	Contains the community name NerveCenter will include in any SNMP v1 or v2c Set request sent to the SNMP Agent on this node. By default this value is "public". This value is not used when doing a Set operation when using SNMP v3.
SNMP v3	
Select User	<p>Selects the configuration to be used when polling with SNMP v3.</p> <ul style="list-style-type: none"> ■ User #1 and User #2 are system-wide parameter sets, configured via NerveCenter Administrator. ■ Local User allows you to enter node-specific credentials that define the parameters for polling the agent on this node. <p>Selecting Local User enables the fields in that area of the SNMP v3 section</p>

Attribute	Description
Security Level	<p>The security level determines whether authentication or encryption services are used in communications between NerveCenter and the node. There are three security levels:</p> <ul style="list-style-type: none"> ■ NoAuthNoPriv: NerveCenter uses neither authentication nor encryption when communicating with the agent, so no passwords are required. ■ AuthNoPriv: Message authentication is used without encryption while communicating with the agent. An authentication protocol and password are required. The authentication password is the same for all nodes managed by the NerveCenter user. ■ AuthPriv: Both authentication and encryption are used when communicating with the agent. Both the authentication and privacy protocols and passwords are required. These passwords are the same for all nodes managed by the NerveCenter user. <p>For more information about SNMPv3 security, see Overview of NerveCenter SNMPv3 Support in Managing NerveCenter. For details about passwords, see Configuring SNMPv3 Security Settings in Managing NerveCenter.</p>
Local User	<p>When Select User is set to Local User, you may enter custom attributes for this node:</p> <ol style="list-style-type: none"> 1. Enter the User Name to use for the node. 2. Configure the Local User Authentication and Privacy as necessary: <ul style="list-style-type: none"> ■ For AuthNoPriv or AuthPriv security, select the Authentication protocol (MD5 or SHA-1) and click Set Auth Key to set the authentication password. ■ For AuthPriv security, you must also select the Privacy protocol (DES, 3DES, AES128, AES192, or AES256) and click Set Priv Key to set the privacy password. 3. Optionally, enter a Context value for the node. (This is empty by default and is only needed for special cases within SNMPv3.)
Engine ID	<p>Holds the SNMP v3 Engine ID for the agent on this node. NerveCenter self-populates this field when establishing communication with the device. However it can be manually set by typing here the value</p>

ICMP Properties

Table 2: Definitions of Node Attributes: ICMP Properties

Attribute	Description
Ping	
Use Defaults	When selected, NerveCenter uses the system-wide defaults when pinging this node. Clear the check box to apply node-specific parameters.

Attribute	Description
IPv4 TTL + IPv6 Hop Limit	Sets how many “hops” the request can proceed across the network towards the node. The default is 128. The range is 0 up to 255.
Payload	Sets the size of the data region, in bytes, that is included in a Ping request. The default is the minimal amount of 56.
IPv4/v6 ICMP Ping Size	Displays the size of the IPv4/v6 ping request given the specified payload.
Polling	
Use Defaults	When selected, NerveCenter uses the system-wide defaults when polling this node. Clear the check box to apply node-specific parameters.
Retry Interval	The delay, in seconds, that NerveCenter will wait for a response to any issued poll request (Get, GetNext, GetBulk) before issuing another request. If Use Default is unchecked, you may enter a value from 1 to 600 (10 minutes).
Attempts	The number of poll requests NerveCenter will issue before declaring a poll to be timed out. If Use Default is unchecked, you may enter a value from 1 to 11 (an initial attempt followed by up to 10 retries).
Timeout	Displays the maximum poll timeout duration (Retry Interval x Attempts).

Trace Properties

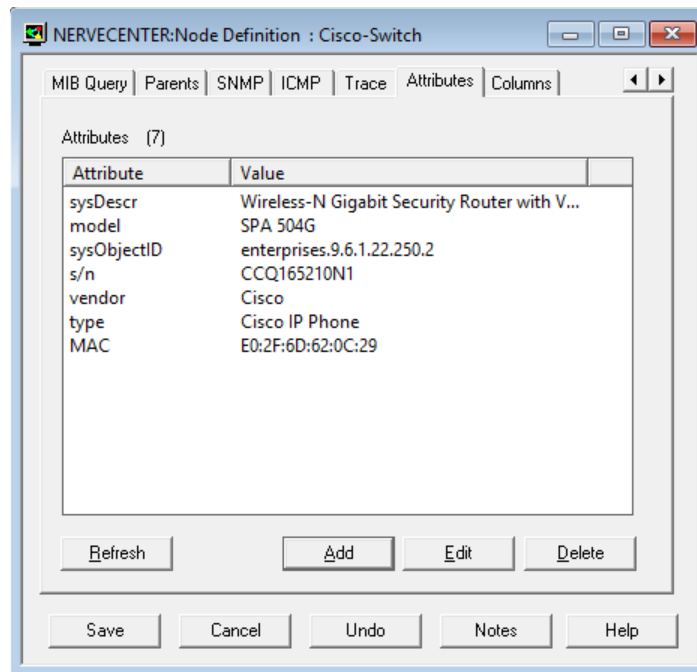
Table 3: Definitions of Node Attributes: Trace Properties

Attribute	Description
Poll Layer Tracing	Select the Trace check box to record polling information for the node. The data is viewable as a spreadsheet, uploaded to the desktop where the NerveCenter Client is running.
Protocol Layer Tracing: SNMP	Select the Trace check box to record SNMP request and response traffic for the node. The data is viewable as a spreadsheet, uploaded to the desktop where the NerveCenter Client is running.
Protocol Layer Tracing: ICMP	Select the Trace check box to record ICMP Ping requests and responses for the node. The data is viewable as a spreadsheet, uploaded to the desktop where the NerveCenter Client is running.

Attributes Properties

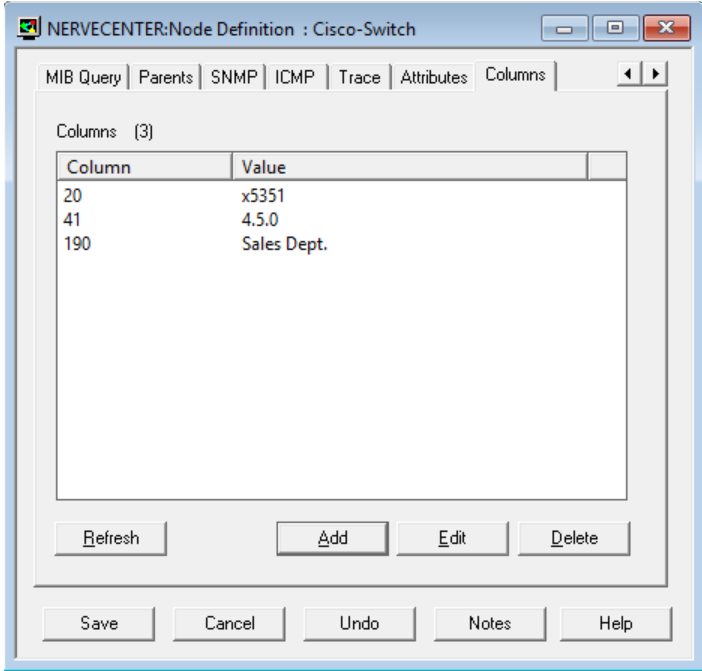
You can add attributes of your own design. Each attribute consists of a name and a value that you specify. Both the name and value of an attribute are text entries and all attributes are saved as part of the node's definition in the NerveCenter node database. There is no limit to how many attributes you create, but the name of each attribute must be unique.

Examples of user-defined attributes could include items such as vendor, model, serial number, config info, access details, usage details and so on.



Columns Properties

You can add attributes of your own design in a second manner. With 'Columns', each attribute consists of an index (a non-negative number) and a value that you specify. The index must be a number and the value is text. All 'Columns' are saved as part of the node's definition in the NerveCenter node database. There is no limit to how many Columns you can add to a node. Each index, though, must be unique.



Property Groups and Properties

Another attribute of a node—one that requires a little explanation—is the node's *property group*. A property group is a list of *properties*, which are strings that generally name either an object in the management information base (MIB) used to manage a node, or the role the node plays in the network (such as “router”). These property strings can be:

- The name of a MIB base object
- A user-defined string

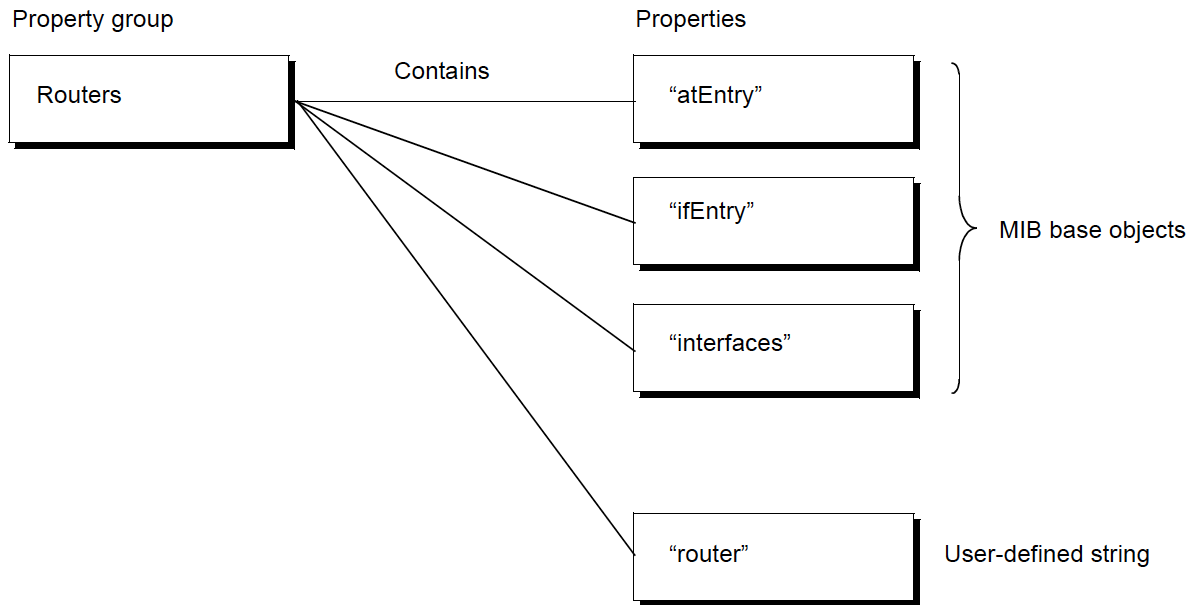


Figure 10: Property Groups and Properties

Property groups are assigned to nodes and control which nodes will be contacted by a particular poll and which nodes can be monitored using a particular alarm. Both types of properties—MIB base objects and user-defined strings—play a part in making these determinations.

For example, NerveCenter ships with a predefined property group called Router. This property group contains the following properties:

- | | | |
|-----------------|---------------------|----------------|
| ■ atEntry | ■ ip | ■ snmp |
| ■ egp | ■ ipAddrEntry | ■ system |
| ■ egpNeighEntry | ■ ipNetToMediaEntry | ■ tcp |
| ■ icmp | ■ ipRouteEntry | ■ tcpConnEntry |
| ■ ifEntry | ■ nl-ping | ■ udp |
| ■ interfaces | ■ router | ■ udpEntry |

In this case, all the properties are MIB objects except “router,” which describes the type of the device.

For the person who programs NerveCenter to monitor particular devices for specific error conditions, the properties associated with each node are important. These properties allow the programmer to define which devices NerveCenter should poll for MIB data and which error conditions NerveCenter should look for on each device, among other things.

You can filter the nodes that you are monitoring based on their properties. For example, you might choose to monitor only nodes that have been assigned the Router property group, that is, all routers.

Polls

A NerveCenter poll periodically sends an SNMP message to a set of nodes, requesting information from the agents running on those nodes. When the poll receives this information from a node, it uses the information in the evaluation of a poll condition, which may cause a trigger to be fired. For example, a poll may fire a trigger if the number of discarded packets on an interface is too high. The poll condition must be able to fire at least one trigger, and may be capable of firing several. These triggers can cause alarms to be instantiated, to change states, or perform actions—under the right circumstances.

The key attributes of a poll are listed in [Table 4](#). This table explains what information these data members contain and, where appropriate, how NerveCenter uses that information.

Poll

Table 4: Definitions of Poll Attributes

Data Member	Definition
Name	A unique name that you assign to the poll.
Property	The Property attribute is a string. This string determines (in part) whether a poll will request MIB data from a particular node. Only if the node’s property group contains the poll’s property can polling possibly occur. However, before a poll will request information from a node’s SNMP agent, other conditions must be satisfied as well. For further information, see the explanation below for Poll Condition.
Port	Optional. If you specify a port number here, NerveCenter will send the poll to this port on the nodes that are configured to receive the poll. Otherwise, NerveCenter will send the poll to the port specified in each node’s definition.
Poll Rate	The number of seconds, minutes, hours, or days between polls.

Data Member	Definition
Overrun Policy	<p>Determines how the Poll should handle an event where it is time to run a poll while the previous instance of that poll is still running. This condition may occur with short Poll Rate settings wherein the connection to the targeted device is returning large numbers of timeouts and retries, or when the data being retrieved is a large table of many thousands of rows. There are four options:</p> <ul style="list-style-type: none"> ■ Skip: Skips this poll and schedules it to occur later per the Poll Rate. ■ Allow: Lets the poll run despite the prior instance still running. ■ Chain: Waits for the prior poll to finish before running this poll. ■ Restart: Cancels the running poll and starts this new poll immediately.
Suppressible	<p>A poll's Suppressible attribute works in conjunction with a node's Suppressed attribute. If a node is suppressed and a related poll is suppressible, that poll will not query that node. If a poll is not suppressible, then it will poll even a suppressed node. Generally, the only polls that are insuppressible are those designed to determine when an unresponsive node becomes responsive again. When a node becomes responsive, the behavior model of which the poll is a part can change the status of the node from suppressed to unsuppressed. (You set an attribute of a node using the Set Attribute alarm action.)</p>
Execute Perl in Global Space	<p>Choose whether the logic of this poll's event handler execute in either the Perl interpreter available to all contexts (Poll, TrapsMasks, PerlSubroutineActions, Action Router Rules) or in a Perl interpreter available only to Poll Handlers.</p>
Handlers	<ul style="list-style-type: none"> ■ On Enabled / Disabled: Select this check box if the Poll requires event handlers which will occur when the Poll is enabled or disabled. (See Enabled attribute.) When selected, additional Property Sheets are added to the Dialog window for containing the OnEnabled and OnDisabled handlers. ■ On Scheduled / Unscheduled: Select this check box if the Poll requires event handlers for when this poll gets initially scheduled for a node or when it is removed from being run against a node. When selected, additional Property Sheets are added to the Dialog window for containing the OnScheduled and OnUnscheduled handlers. ■ On Start / End Execution: Select this check box if the Poll requires event handlers that are called when each run of this poll against a node are started or ended. When selected, additional Property Sheets are added to the Dialog window for containing the OnStart and OnEnd handlers.
Enabled	<p>A poll's enabled status (Off or On) is similar to a node's Managed status. That is, if a poll is disabled, it will never send a request to an SNMP agent.</p>

Trace

Select the **Trace** check box to record basic poll execution data to a spreadsheet. The spreadsheet can be uploaded to the desktop running Client by using the View Log button.

Trigger

If a poll fires a trigger, that trigger has the attributes shown in [Table 5](#).

Table 5: Definitions of Trigger Attributes

Data Member	Definition
Name	The name of the trigger, which is defined in the poll definition.
Node name/IP address	The name or IP address of the node that responded to the poll and caused the trigger to be fired.
Subobject	In general, the Subobject has a value of the form <i>BaseObject.Instance</i> . <i>BaseObject</i> is the name of the MIB base object that the poll inquired about, and <i>Instance</i> is the unique identifier associated with a row of MIB data returned by the poll. In most cases, <i>Instance</i> is the number associated with a particular interface on the node. The subobject, however, can also be an arbitrary string. The important thing is that subobjects can be used to uniquely identify alarm instances so that triggers can be directed to exactly the right alarm instance.
Property	The Property, as always, is simply a string. A trigger fired by a poll does not have a property, but as you'll see later, other triggers do.
Variable bindings	The trigger also contains the values of the MIB attributes referred to in the Poll Condition. Each attribute and its value are called a <i>variable binding</i> .

A trigger's Name, Node name/IP address, Subobject, and Property are all important when it comes to determining what effect, if any, a trigger has on an alarm. You'll find more on this subject in the section [Constructing Behavior Models on page 47](#).

Trap Masks

A trap mask filters SNMP traps that NerveCenter receives. Based on criteria that you specify, the trap mask either filters out each trap or fires a trigger in response to it. A trigger fired by a mask is exactly the same as a trigger fired by a poll except that a trap trigger contains the trap's variable binding list instead of the values of MIB attributes. (For further information about the trigger object, see the section [Polls on page 37](#).)

The principal attributes of a trap mask are shown in [Table 6](#). The table explains what information these data members contain and, where appropriate, how NerveCenter uses that information.

Note: The names of the data members shown above match the labels used in NerveCenter's Mask Definition window, where you create and modify trap masks.

Table 6: Definitions of Trap Mask Attributes

Attribute	Definition
Name	The name of the trap mask.
Generic	<p>The generic trap type is an enumeration constant indicating the nature of the event being reported:</p> <ul style="list-style-type: none"> ■ 0—coldStart ■ 1—warmStart ■ 2—linkDown ■ 3—linkUp ■ 4—authenticationFailure ■ 5—egpNeighborLoss ■ 6—enterpriseSpecific <p>You supply a Specific trap number (see below) only if the generic trap type is 6.</p>
From	Indicates that the object identifier (OID) contained in the trap's Enterprise field must represent a branch in the MIB tree that is the same as, or subordinate to, the branch represented by the contents of the trap mask's Enterprise field.
From Only	Indicates that the OID contained in the trap's Enterprise field must <i>match</i> the trap mask's Enterprise attribute exactly.
Enterprise	An OID (or name) representing the object referenced by the trap.
Specific	A trap number supplied by the vendor of the product whose agent generated the trap. The significance of the trap number is defined in an ASN.1 file provided by the vendor.
Trigger Type	Trigger Type can be set to either Simple Trigger or Trigger Function. See the next two table entries for definitions of these trigger types.
Simple Trigger	<p>A simple trigger is one that will be fired whenever the trap mask sees a trap that meets the criteria specified in the fields discussed above.</p> <p>The value of this attribute affects how this object interacts with other objects in a behavior model.</p>
Trigger Function	<p>A trigger function is a Perl script that is called whenever the trap mask sees a trap that meets the criteria specified in the fields discussed above. This function typically looks at information in the trap's variable bindings and fires a trigger if a condition is fulfilled. The trigger function fires this trigger using NerveCenter's FireTrigger() function.</p> <p>The value of this attribute affects how this object interacts with other objects in a behavior model.</p>
Enabled	As with a poll, a trap that is disabled (Enabled is set to Off) is nonfunctional.

Alarms

As mentioned in the section [Behavior Models on page 17](#), a NerveCenter alarm consists primarily of a state diagram, which defines the alarm's states, the transitions between states, and the alarm actions to be performed when each transition takes place. This alarm definition is analogous to a class in object-oriented programming. That is, the alarm itself does not monitor a network condition; rather, an alarm instance (comparable to an object) is created to track such a condition.

For example, the section [Behavior Models on page 17](#) showed the definition of an alarm designed to monitor traffic on an interface.

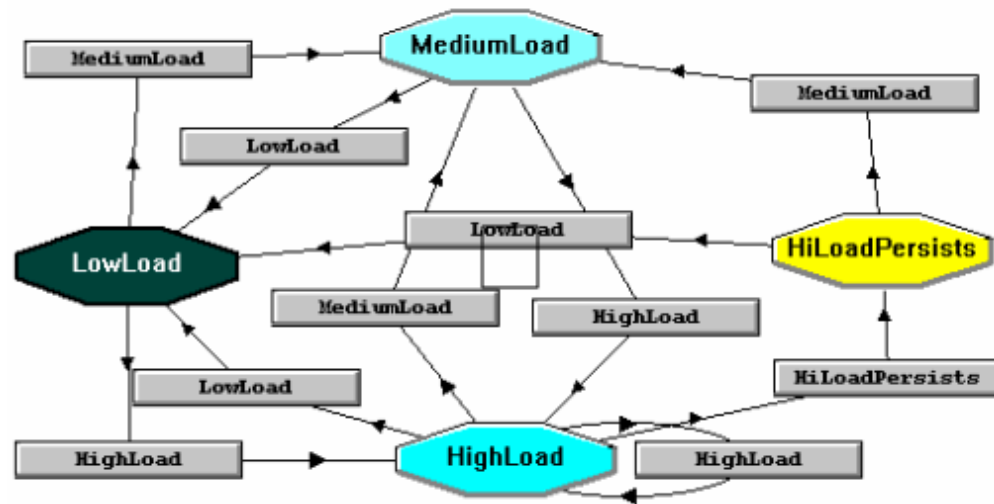


Figure 11: Definition of the alarm IfLoad

If NerveCenter detects a medium or high level of traffic on an interface it is managing, it creates an instance of this alarm to track the condition. If NerveCenter detects medium or high traffic on five interfaces, it creates five instances of the alarm. Each instance of the alarm maintains such information as:

- The instance's current state
- The severity of that state
- The node the instance is monitoring

In addition, each alarm instance causes the appropriate alarm actions to take place when a state transition occurs.

If five instances of IfLoad are created, how do you distinguish them? Depending on the *scope* of the alarm, you might need to look at the instance's *node* attribute or at both its *node* and *subject* attributes.

In NerveCenter, alarms can have one of four scopes: enterprise, instance, node, or subject. Only one instance of an enterprise-scope alarm can be created. This instance monitors a condition across all managed nodes. For example, one alarm instance could cause an action to take place if three or more routers in an enterprise are down at the same time.

A node-scope alarm monitors a single managed device for a condition. For instance, the alarm `SnmpStatus` (shipped with NerveCenter) determines whether a device is in a normal state, unreachable, down, or up but unable to respond to SNMP requests. An instance of this type of alarm can be identified by its alarm name and the name of the node it is monitoring. This node name is an attribute of the alarm instance.

A subobject-scope alarm most often monitors an interface on a device. For example, an instance of the alarm `IfLoad` monitors each interface that is experiencing a medium to high level of traffic. This type of instance can be identified by its alarm name, the name of the node it is monitoring, and the name of the subobject being monitored. This subobject name is usually composed of the name of a MIB table followed by an instance number. That is, if an instance of the `IfLoad` alarm is monitoring port 2 on a device, its subobject attribute has the value `ifEntry.2`.

Instance scope alarms track instances for every interface or port that fits the polled condition regardless of the base object. Instance scope is similar to Subobject scope but has the following difference: Instance scope lets you monitor any instance for different base objects. This allows you to track a variety of events for any managed subobject in a single alarm instance.

Alarm Scope

All NerveCenter alarms have a property called *scope*. This property can have one of four values:

- Subobject
- Instance
- Node
- Enterprise

If an alarm has Subobject scope, an instance of that alarm tracks activity on a component that can be described using a nonzero MIB object instance, for example, an interface on a router.

Instance scope alarms track instances for every interface or port that fits the polled condition regardless of the base object. Instance scope is similar to Subobject scope but has the following difference: Instance scope lets you monitor any instance for different base objects. This allows you to track a variety of events for any managed subobject in a single alarm instance.

If an alarm has Node scope, an instance of that alarm tracks activity on a single device. If an alarm has Enterprise scope, an instance of that alarm tracks activity on all managed nodes.

Note: It might be useful to think of an alarm instance as a copy of the alarm's state diagram whose current state is something other than Ground.

Why is NerveCenter architected this way? Well, think about the following network management problem: You want to be notified whenever four interfaces on a device experience high traffic.

Your first step in solving this problem might be to create a poll that detects high traffic on an interface and fires the trigger `highTraffic`. You might then create an alarm with node scope and five states, as shown in Figure 12.

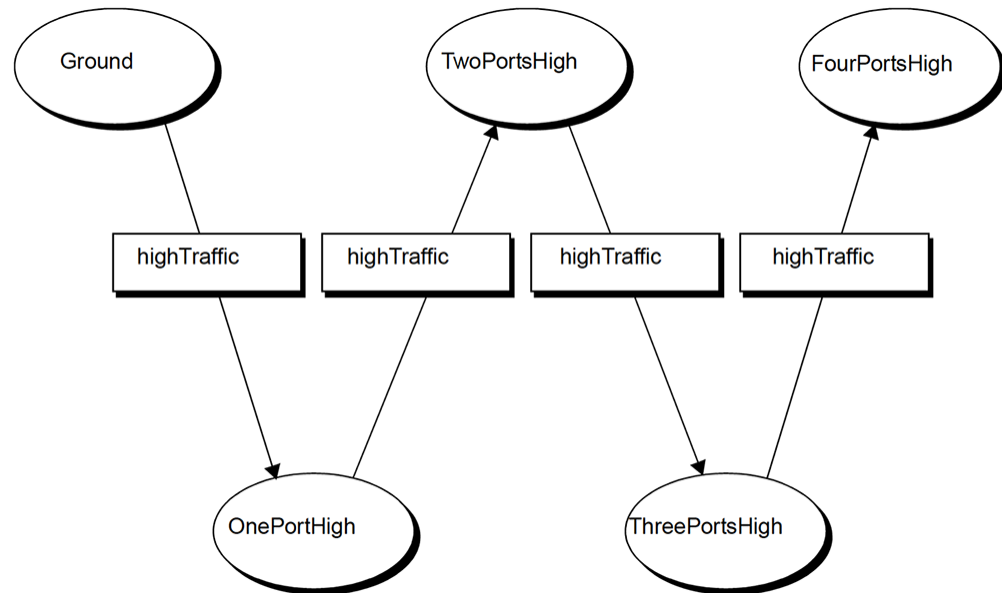


Figure 12: Possible Alarm Diagram for Looking for High Traffic on Four Interfaces

Most likely, this alarm won't detect the condition you're looking for because all four transitions can be effected if the poll repeatedly detects high traffic on a *single* port.

To solve your problem, the trigger `highTraffic` must cause one or more transitions in a *subject* scope alarm, and this alarm must fire a `busyPort` trigger (using the Fire Trigger alarm action) during its final transition. Such an alarm is shown in Figure 13.

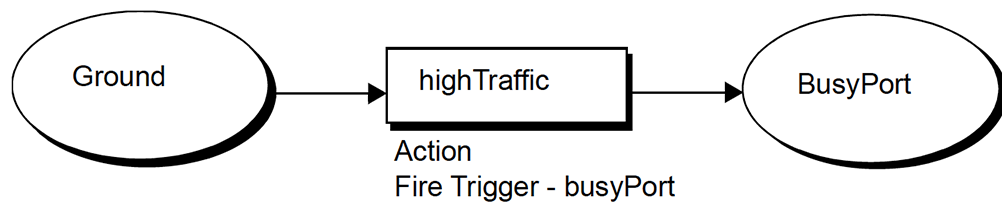


Figure 13: A Subobject Scope Alarm

When the high-traffic poll detects high traffic on an interface, a subobject scope alarm will be instantiated, and the transition `highTraffic` will occur. During this transition, the alarm will fire a trigger called `busyPort`. Note that once a subobject alarm instance transitions to the `BusyPort` state, additional high-traffic triggers for the interface concerned have no effect. However, if the high-traffic poll detects high traffic on other interfaces, new alarms will be instantiated and fire the trigger `busyPort`. Each instance fires its own `busyPort` trigger.

Now a node scope alarm similar to the one shown in [Figure 14](#) can be configured to receive up to four `busyPort` triggers, each one from its own instance of the high traffic alarm. Each `busyPort` trigger signals high traffic on a different interface.

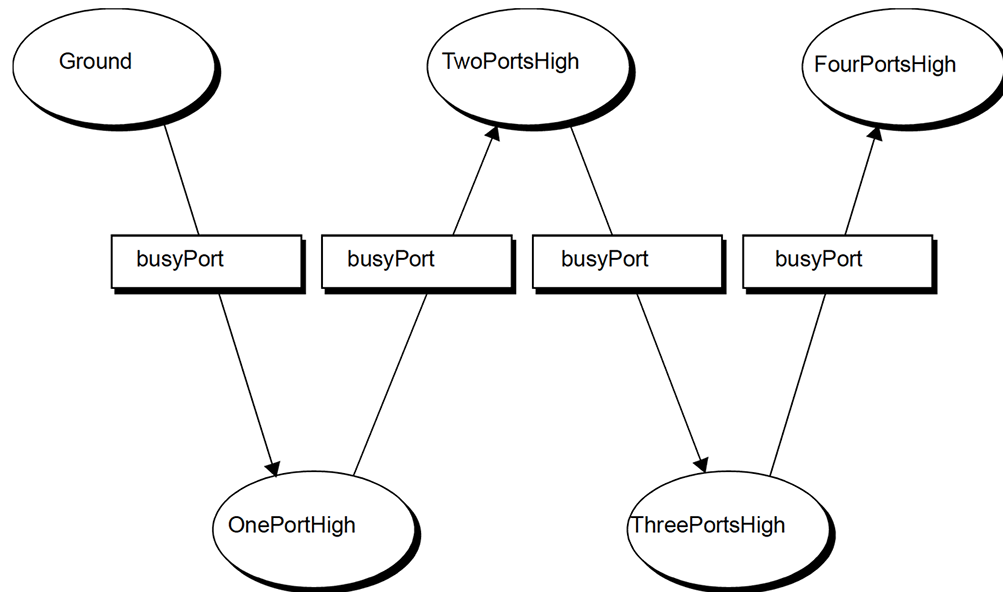


Figure 14: Node Scope Alarm Detecting High Traffic from Four Alarm Instances

An instance scope alarm behaves in a similar manner as the subobject scope alarm. The main difference between subobject and instance scope is that, with instance scope, you could add another transition to the alarm to monitor a different base object than the one for high traffic. Then, the alarm could be instantiated by the high-traffic poll and then transition again when an entirely different condition (MIB object) is detected.

NerveCenter and Perl

In prior versions of NerveCenter, there was one Perl interpreter and that interpreter was single threaded. This meant that only one poll, trap mask function, Perl subroutine, or action router rule could run at one time. Perl scripts that take a long time to run, such as logging to a file, performing database queries, or issuing external system calls, can slow down NerveCenter's performance. As shown in [Figure 15](#), all poll conditions, trigger functions, action router rules and Perl subroutines all used the same Perl interpreter.

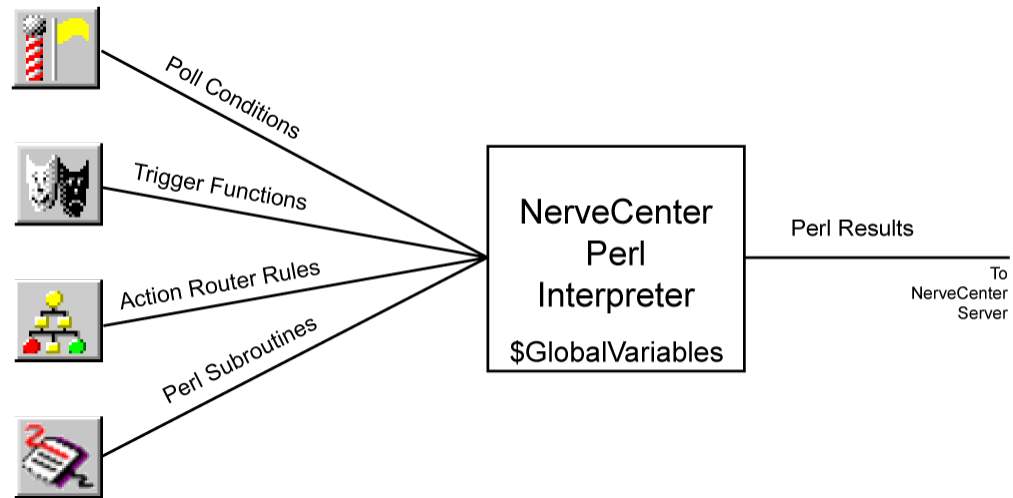


Figure 15: Previous NerveCenter Perl Interpreter Architecture

The advantage of this architecture is that you can use the same variables through out your Perl in NerveCenter. The disadvantage comes with high-use situations. Since there is only one Perl interpreter, only one Perl routine can run at a time. If your NerveCenter installation is receiving thousands of traps, and hundreds of these traps cause Perl-intensive triggers to fire or polls to run, with hundreds of Perl subroutines to follow as actions, NerveCenter performance can quickly degrade.

NerveCenter today gives you the option of using separate Perl interpreters for the three major components in NerveCenter which use Perl—poll conditions, trigger functions, perl subroutines. Action router rules and OpC trigger functions continue to use a global—or shared—Perl interpreter. Figure 16 shows the new architecture.

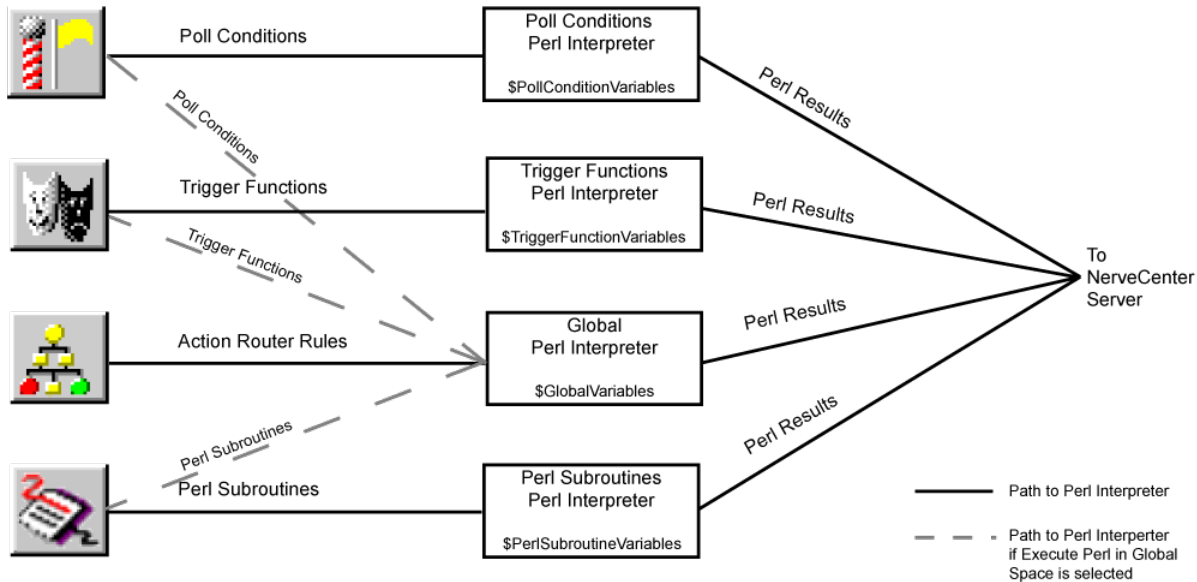


Figure 16: CurrentNerveCenter Perl Interpreter Architecture

With four different interpreters handling the Perl work load, less time is spent waiting for one Perl routine to finish. Polls can run independent of triggers or Perl subroutines. Remember, now that each interpreter is separate, global variables only work within one interpreter.

Using multiple interpreters is optional. If it is better for your NerveCenter configuration to continue using a single interpreter, you can choose to send some or all of your poll conditions, trigger functions, and Perl subroutines to the Global Perl interpreter used by the action router and OpC triggers.

Constructing Behavior Models

Given the NerveCenter objects discussed in [NerveCenter Objects on page 30](#), it's possible to create a *behavior model*, which can be defined as the set of NerveCenter objects required to deal with a single network or system condition. [Figure 17](#) shows a simple example of the objects that might make up a behavior model.

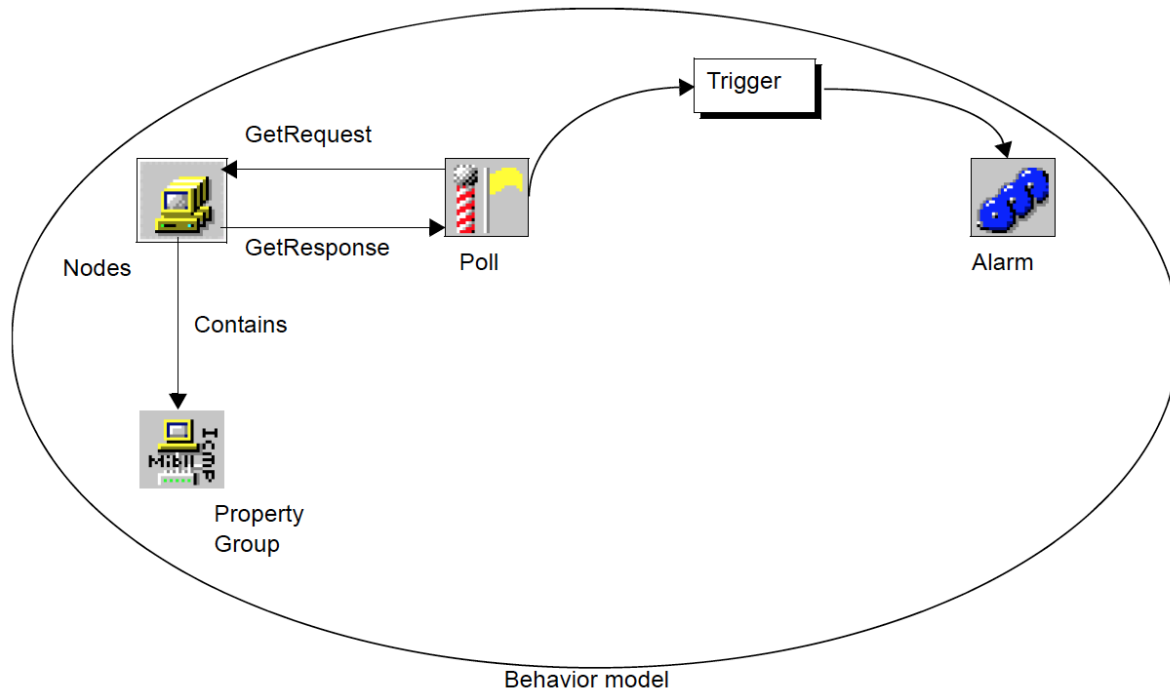


Figure 17: A Behavior Model

The next two sections:

- Discuss in general how the various objects fit together to make a model
- Present an example of a behavior model

How the Pieces Fit Together

Let's first review how you define which managed nodes a behavior model will monitor and manage. As Figure 18 shows, each node belongs to a property group, and that property group contains properties.

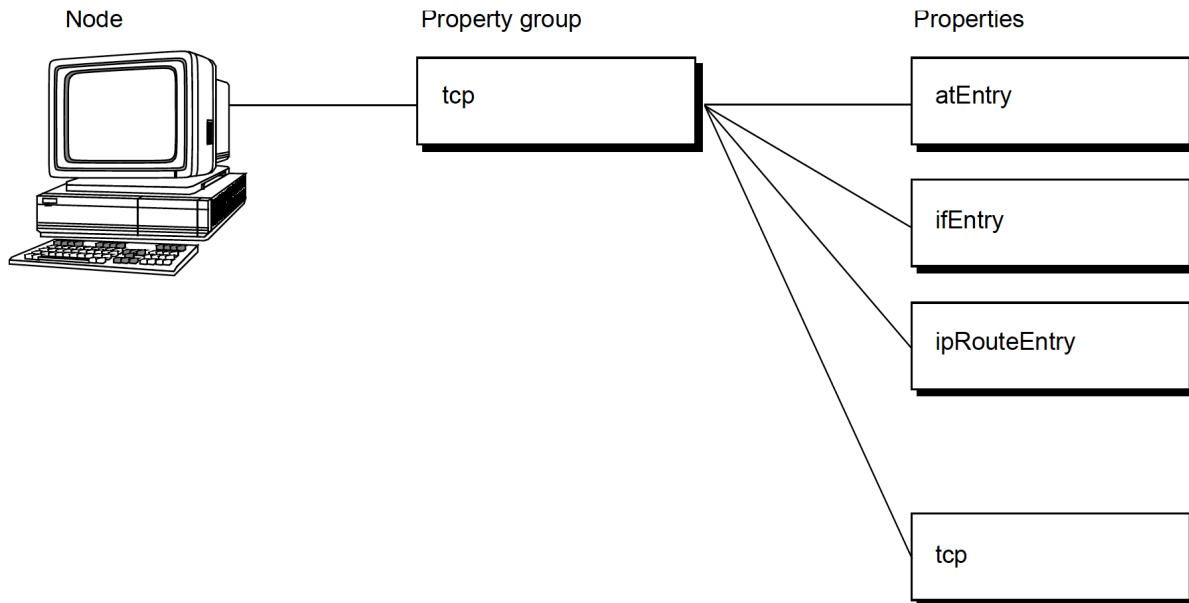


Figure 18: Nodes, Property Groups, and Properties

Any set of nodes that share a unique property can be managed as a set of devices. (The nodes need not be members of the same property group.) In the figure above, the tcp property might be that unique property.

For a node to be pollable, the principal requirements are that:

- The poll's property must be in the node's property group.
- The base object around which the poll's poll condition is built must be a property in the node's property group.
- The poll's trigger must correspond to a pending alarm transition, and the alarm's property must be in the node's property group.

Figure 19 shows the definition of a poll that has been designed to work with the node shown in Figure 18.

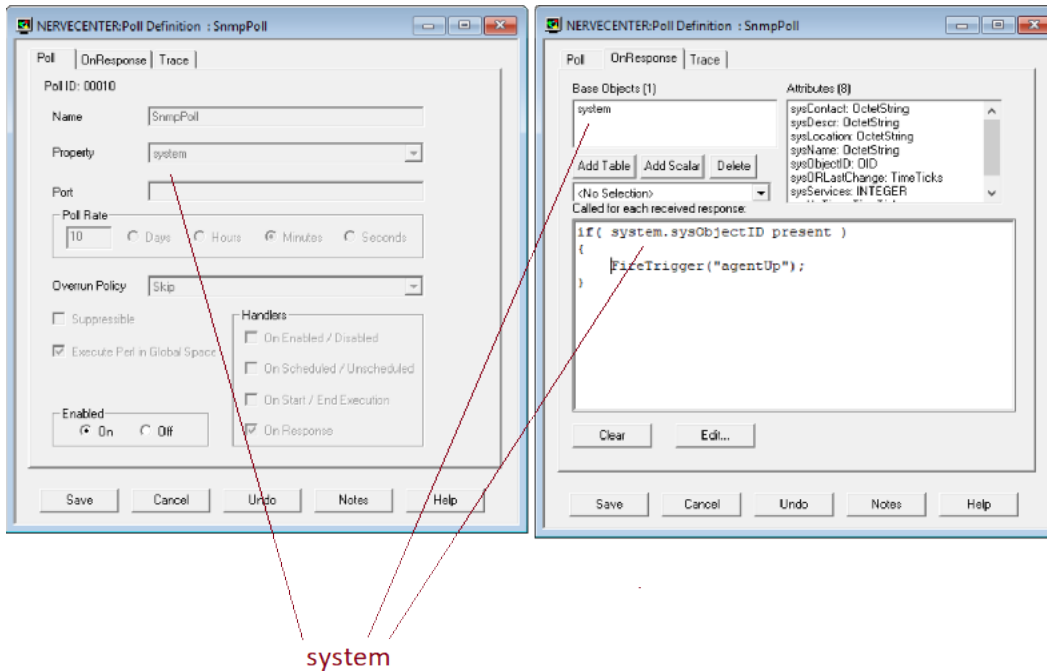


Figure 19: Relationship Between Node and Poll

As you can see, the node's property group, Mib-II, contains a property tcp that matches the poll's property *and* the base object used in the poll's poll condition. Once this poll is enabled, the poll TcpMedRetrans *will* poll the node, unless there is no alarm that the poll can affect or the node is suppressed. (If the node is suppressed, no polling will occur because the poll is marked suppressible.)

Note: Since trap masks do not have properties, this type of matching is not necessary for masks.

If TcpMedRetrans polls the node, receives a response to its query, and that response satisfies the poll condition, the poll will fire a trigger. If an alarm has been defined whose first transition is tcpRetransMed (the poll's trigger) and that alarm is enabled has the property tcp, a new instance of that alarm will be instantiated to monitor the node. Because the alarm is instantiated using the trigger's Node and Subobject, the key attributes of the trigger and alarm will match, and the first transition will be effected.

Once an alarm instance has been instantiated and has gone through one transition, the transitions that can be effected from its current state determine which triggers affect the alarm. For example consider the following alarm, TcpRetransMon.

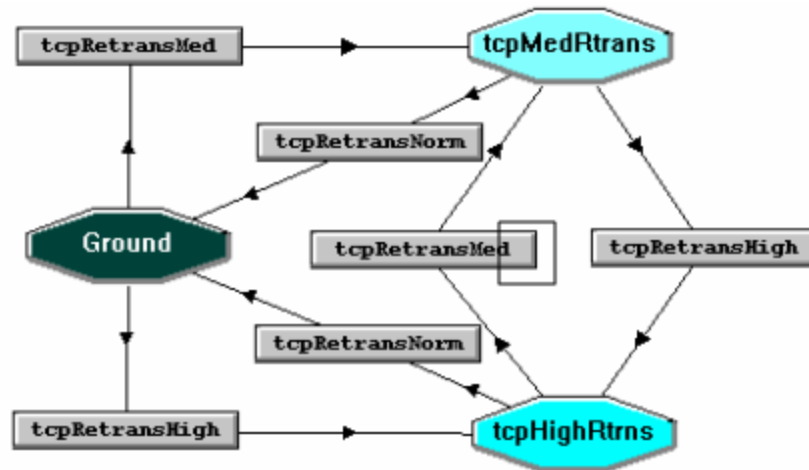


Figure 20: An Alarm: TcpRetransMon

When this alarm is first instantiated and the tcpRetransMed transition is made, the alarm transitions to the tcpMedRtrns state, so two transitions are pending: tcpRetransNorm and tcpRetransHigh. If NerveCenter sees a trigger with one of those names, and the trigger's Node and Subobject match those of the transition, the transition occurs.

An Example of a Behavior Model

This section presents an overview of the set of steps you would need to perform to create a behavior model that monitors node interfaces. The possible interface conditions are link up and link down.

Note: Don't try to follow these directions. Just read over them to get an overview of the procedure. Detailed procedures are available in following chapters.

1. Create a property group named CheckLink.
2. Add to this property group the properties ifEntry (base object) and checkLink (user defined).
3. Assign the property group CheckLink to all of the managed nodes whose interfaces you want to monitor.

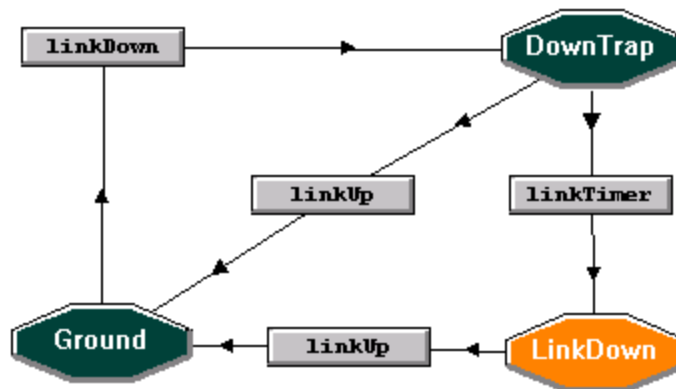
4. Create two masks: LinkUp and LinkDown.
The values you use to create LinkUp are shown in the table below.

Table 7: Values Needed to Create LinkUp

Attribute	Value
Name	LinkUp
Generic	LinkUp=3
Trigger Type	Simple Trigger
Enabled	On

The definition for LinkDown is the same as the definition of LinkUp except for the name of the mask and Generic SNMP trap number (LinkDown=2).

5. Create the alarm shown below.



Once this alarm is enabled, the behavior model will become functional.

The IfLinkUpDown alarm contains the property ifEntry, which is in the property group CheckLink. Even though a trap mask filters all traps sent to NerveCenter, the IfLinkUpDown alarm will only become instantiated when the SNMP agent sending the trap belongs to a node in the CheckLink property group.

Here's how the behavior model might interact with one port on a workstation that belongs to the property group:

1. The mask LinkDown will cause a transition to the DownTrap state, as well as start a three-minute timer (linkTimer).
2. If the agent comes back up, then the alarm transitions back to Ground and the timer is cleared.
3. If three minutes has passed and the interface remains down, then the alarm transitions to LinkDown and sends a 7004 Inform to the network management platform.

Before you can begin monitoring your network using the NerveCenter Client, you must start the client and then establish a connection between the client and one or more NerveCenter servers. You may also want to set up alarm filters to control which alarm instances the NerveCenter Client will display information about.

Starting the NerveCenter Client

TO START THE NERVECENTER CLIENT

- Select the **Start** menu. Select **Programs > LogMatrix NerveCenter > Client**.
NerveCenter displays the NerveCenter Client window.

Note: These steps depend on a typical NerveCenter installation. The directory path may be different.

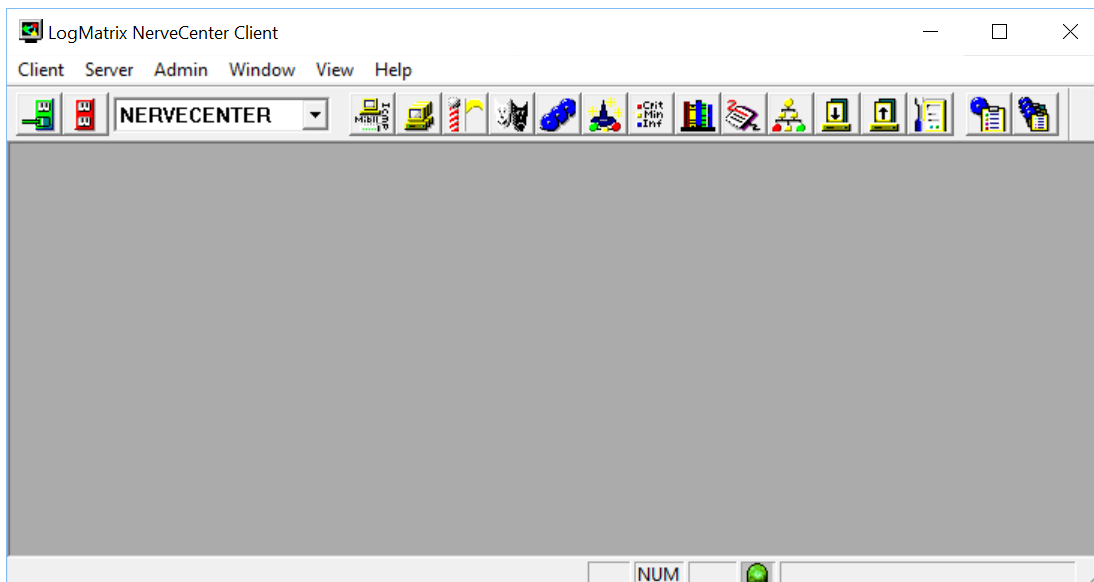


Figure 21: The NerveCenter Client Window

Most buttons and menu options are not enabled until you connect the client to a NerveCenter server.

Connecting to a Server

Before you can use the client, you must connect the client to a NerveCenter server. This server collects data from managed devices, creates alarm instances, and performs the actions defined in alarms. The server also gives the client access to information about alarm instances and the status of nodes.

You can connect your client to more than one server at one time and view information about all the active alarm instances being managed by those servers. However, only one server can be the *active* server. The active server determines which NerveCenter database is used when you ask for a list of polls or the definition of an alarm.

For information on connecting to a NerveCenter server, see the following subsections:

- [Connecting to a Server Manually on the facing page](#)
- [Connecting to a Server Automatically on page 57](#)
- [Sharing MIB Information from Multiple Servers on page 59](#)

You may also be interested in the following topics, which relate to connecting to a server:

- [Selecting the Active Server on page 60](#)
- [Deleting a Server from the Server List on page 60](#)
- [Changing the Server Port on the Client on page 61](#)

Connecting to a Server Manually

If you haven't configured the client to connect to one or more servers at startup, or if you want to establish a connection with a server that you don't typically use, you must establish your connection with the server manually.

TO CONNECT TO A NERVECENTER SERVER MANUALLY

1. From the **Server** menu, select **Connect**.

The Connect to Server window displays.

2. In the **Server Name** field, type the hostname or IP address of the machine where the NerveCenter server is running or select a hostname or IP address from the **Server Name** drop-down list.

The first time you connect to a server, the drop-down list is empty. After that, it contains a list of the machines to which you've connected, or attempted to connect, in the past. (The list won't display the names of machines to which you're already connected.) For information on removing an entry from the drop-down list box, see the section [Deleting a Server from the Server List on page 60](#).

3. Type a user name and password in the **User ID** and **Password** fields.

You must enter a user name and password. The user whose name you enter here must be a member of the NerveCenter Users (ncusers) or NerveCenter Admins (ncadmins) group on the NerveCenter Server host.

4. Select the **Connect** button.

If the machine to which you try to connect is not running the NerveCenter server, you see the message **The server did not respond**.

When the client successfully connects to the server, all of the buttons in the button bar become enabled, and the Aggregate Alarm Summary window appears.

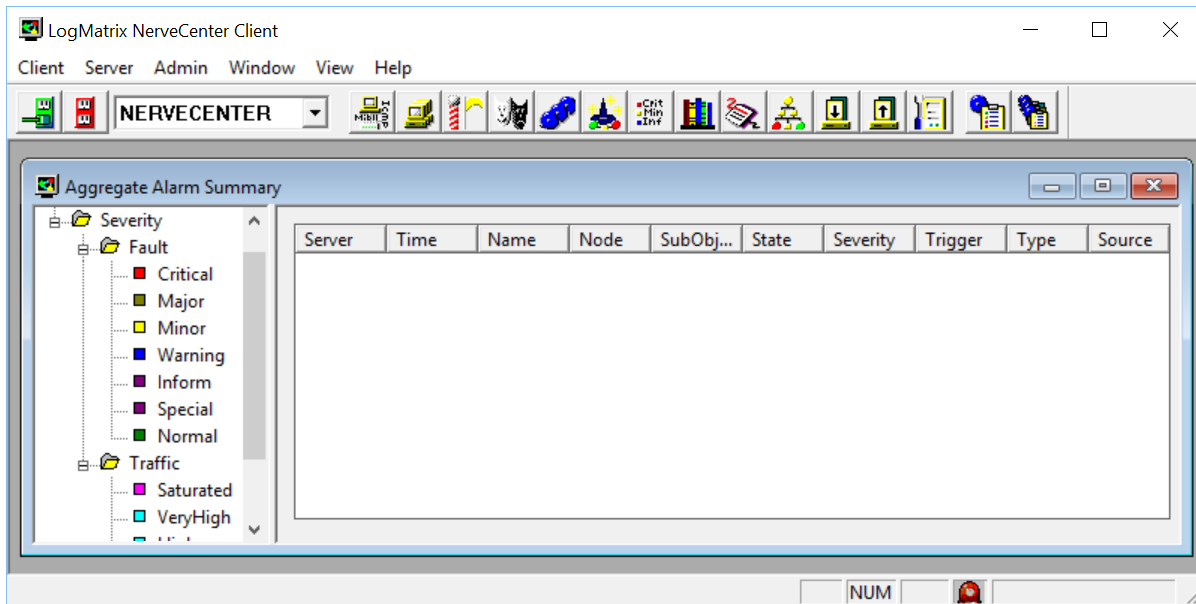


















Figure 22: Client Connected to a Server

Table 8 lists the client windows you can reach by using the buttons in the client's toolbar.

Table 8: Windows Accessible from Toolbar

Button	Window
	Opens the Connect to Server window, from which you can connect the client to a NerveCenter server.
	Opens a Client message window containing the prompt Disconnecting from Hostname . Use this window to confirm that you want to disconnect the client from a NerveCenter server.
	Opens the Property Group List window. From this window, you can view the currently defined property groups and the properties that each property group contains.
	Opens the Node List window. From this window, you can view a list of the nodes defined in the NerveCenter database and a brief definition of each node.
	Opens the Poll List window. From this window, you can view a list of the polls defined in the NerveCenter database and a brief definition of each poll.
	Opens the Mask List window. From this window, you can view a list of the trap masks defined in the NerveCenter database and a brief definition of each trap mask.
	Opens the Alarm Definition List window. From this window, you can view a list of the alarms defined in the NerveCenter database and open a definition window for each alarm.
	Displays a list of currently defined correlation expressions. Correlation expressions enable you to create alarms from Boolean expressions.

Button	Window
	Opens the Severity List window, from which you can view a list of the severities defined in the NerveCenter database. (A severity has a name, a severity level, and a color associated with it.)
	Opens the Perl Subroutine List window. From this window, you can view a list of the currently defined Perl subroutines.
	Opens the Action Router Rule List window. From this window, you can view a list of the current set of rules that you have defined for the Action Router.
	Opens the Import Objects and Nodes dialog. From this dialog, you can import behavior models from one NerveCenter to another.
	Opens the Export Objects and Nodes dialog. From this dialog, you can export specific objects or groups of objects from one database to another.
	Opens the Server Status dialog. This dialog provides you with a comprehensive view of all your NerveCenter server settings.
	Opens the Alarm Summary window. This window presents information about the current alarm instances for the active server.
	Opens the Aggregate Summary window. This window presents information about the current alarm instances for all the servers to which you're connected.

Connecting to a Server Automatically

If you want to establish a connection with the same set of servers each time you run the client, you can use NerveCenter's Autoconnect feature.

Note: Before you activate the Autoconnect feature, you might want to manually connect to the NerveCenter Server, to verify that you can indeed access the server.

TO SET UP A LIST OF SERVERS TO WHICH YOU'LL CONNECT AT STARTUP

1. From the client's **Client** menu, choose **Configuration**.

The Client Configuration dialog displays.

2. Enter the hostname or IP address of the server to which you want to connect in the **Server Name** field.
3. Generally, you'll leave the default value in the **Server Port** field.
However, if the administrator who configured the server you want to connect to has changed the server port to be used for client/server communication, you must enter the new port number here. The NerveCenter Client uses this same port number for every NerveCenter Server to which it attempts to connect.
4. Check the **Autoconnect** checkbox.
5. Type a user name and password in the **User ID** and **Password** fields.
You must enter a user name and password. The user whose name you enter here must be a member of the NerveCenter Users (ncusers) or NerveCenter Admins group (ncadmins) on the NerveCenter Server host.
6. Select the **Add** button.
The server's name and automatic-connection status are displayed in the list near the bottom of the window.
7. Repeat the process for each server you want to connect to automatically.

8. Select the **OK** button.

When you restart and log on to the client, you will be connected to the servers that have an Autoconnect status of On. Alternatively, you can connect, or reconnect, to these servers by selecting **Autoconnect** from the client's **Server** menu.

Sharing MIB Information from Multiple Servers

The NerveCenter Client needs a copy of the same MIB file that a NerveCenter Server uses to provide MIB base objects and attributes. If you intend to connect to multiple servers that use the same MIB file, you can direct NerveCenter to share MIB information. When you use this option, the NerveCenter Client saves only the MIB information sent to it by the first connected server.

For more information about MIBs, refer to [Managing Management Information Bases \(MIBs\)](#) in *Managing NerveCenter*.

TO SHARE MIB INFORMATION

1. Disconnect from any connected servers.
2. From the client's **Client** menu, choose **Configuration**.

The Client Configuration dialog is displayed.

The screenshot shows the 'Client Configuration' dialog box with the following details:

- Server Connections** tab is selected.
- Connection Information** section:
 - Server Name: NERVECENTER
 - User ID: nccadmin
 - Server Port: 32504
 - Password: [REDACTED]
 - Autoconnect
- Buttons:** Add, Update, Delete
- Server List** table:

Name	Autoconnect
NERVECENTER	Off
NC6200-CENTOS6-G	Off
- Server Port:** 32504
- Heartbeat Configuration:**
 - Heartbeat
 - Retry Interval (sec): 30
- Share MIB (client uses MIB from first connected server)
- Buttons:** OK, Cancel, Help

3. Select the **Share MIB** checkbox.

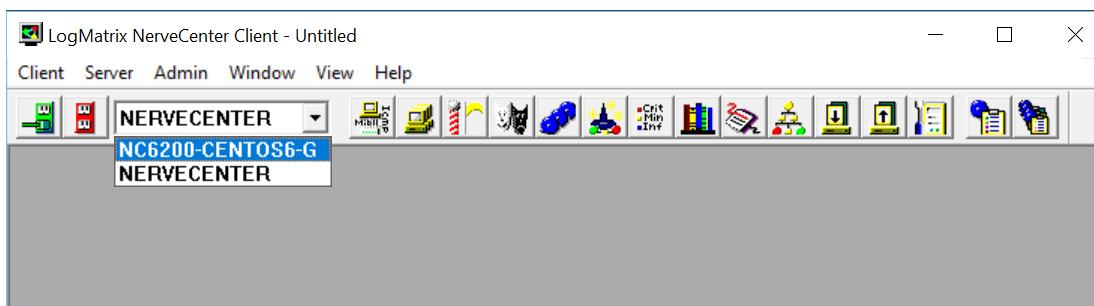
4. Select the **OK** button.

Selecting the Active Server

The active server is the one whose database you can read data from. That is, you have access to this server's alarm definitions, poll definitions, and so on. You can view alarm instances for any number of servers at the same time.

TO MAKE A PARTICULAR SERVER THE ACTIVE SERVER

1. Display the server drop-down list on the client's button bar.



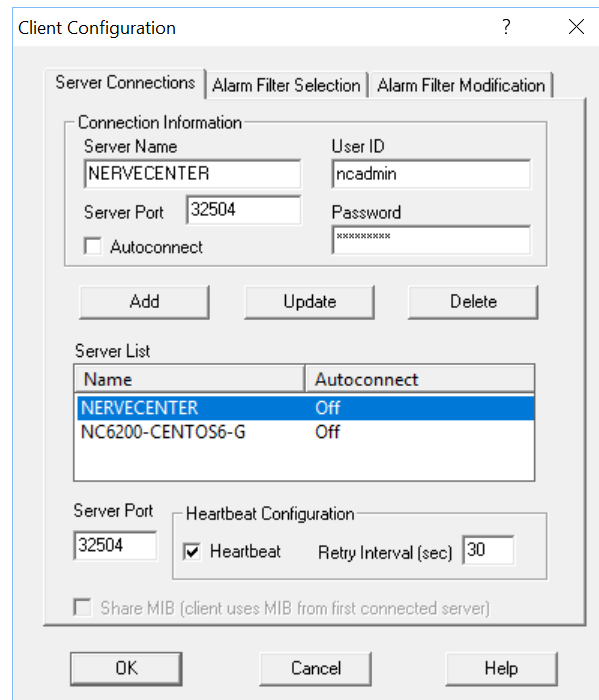
2. Select from the list the name of the server you want to make the active server.
The name of the active server appears in the drop-down list box.

Deleting a Server from the Server List

NerveCenter maintains a list of servers that a client has connected to, or attempted to connect to, in the past. This list is used in the Connect to Server window, which you use to establish a connection to a server manually, and it also appears in the Client Configuration window. This list may contain the names of servers that you will never connect to again, or, even worse, the misspelled names of servers you were unable to connect to because of a misspelling.

TO DELETE THE NAME OF A SERVER FROM THE SERVER LIST

1. From the client's **Client** menu, select **Configuration**.
NerveCenter's Client Configuration window is displayed.



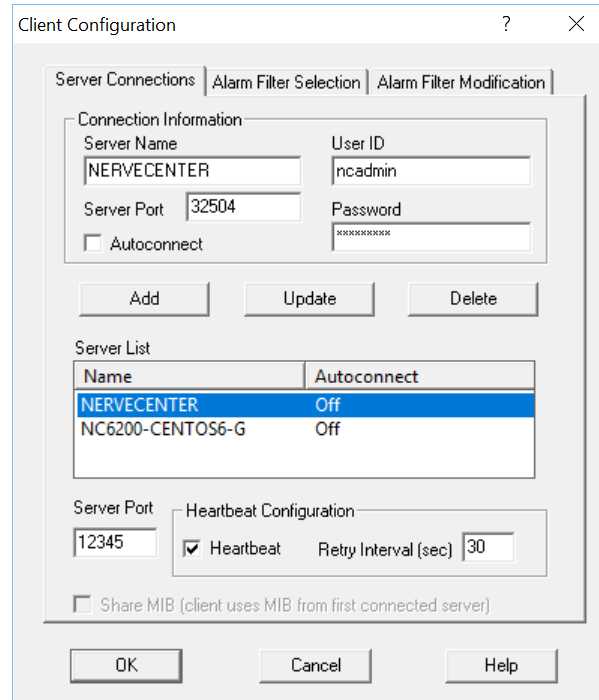
2. In the **Server List** near the bottom of the window, select the server name you want to remove from the server list.
3. Select the **Delete** button.
4. Select the **OK** button.

Changing the Server Port on the Client

Each NerveCenter server uses a special port on its host for client/server communication. By default, servers use port 32504; however, the person who configures the NerveCenter server can change the number of this communication port if port 32504 is being used by another application. If this number is changed on the server side, you must make a corresponding change on the client side before you will be able to connect to the server.

TO CHANGE THE CLIENT'S SERVER PORT

1. From the client's **Client** menu, choose **Configuration**.
The Client Configuration window is displayed.



2. In the **Server List** near the bottom of the window, select the name of the server that uses the non-default port number.
Connection information for that server is displayed.
3. Type the new port number in the **Server Port** text field.
4. Select the **OK** button.

Setting Up Alarm-Instance Filters

Before or after you've connected to the servers from which you want to retrieve alarm instances, you can set up one or more alarm-instance filters, per server. These filters control which alarm instances are displayed in the NerveCenter Client. You can filter alarm instances by:

- The IP address of the instance's node
- The severity of the instance's state
- The property group associated with the instance's node

If you filter alarm instances by a severity, only instances whose states have this severity will be displayed in the client. Filters based on property groups and IP address ranges work similarly.

A single filter can contain any combination of:

- A list of subnets
- A list of severities
- A list of property groups

These filters offer two advantages. First, they limit the number of alarm instances that will show up in the client, enabling you to focus your attention on the alarm instances that are specifically of interest to you. Using filters also improves the performance of the client, since NerveCenter only transfers to the client those alarm instances that match the filter criteria.

For information on how to build an alarm-instance filter and on how to associate a filter with a server, see the sections listed below:

- [Filtering Alarms by IP Range below](#)
- [Filtering Alarms by Severity on page 70](#)
- [Filtering Alarms by Property Groups on page 74](#)
- [Associating a Filter with a Server on page 77](#)
- [Rules for Associating Filters with Alarms on page 79](#)

Filtering Alarms by IP Range

When you filter alarms by IP range, you are specifying that you only want to display alarm instances in the NerveCenter Client from particular nodes identified by their IP addresses. See [IP Subnet Filter Exclusion Rules on page 66](#), for more about filtering alarms by IP ranges. Although you can create a filter simply based on an IP range, a single filter can contain any combination of:

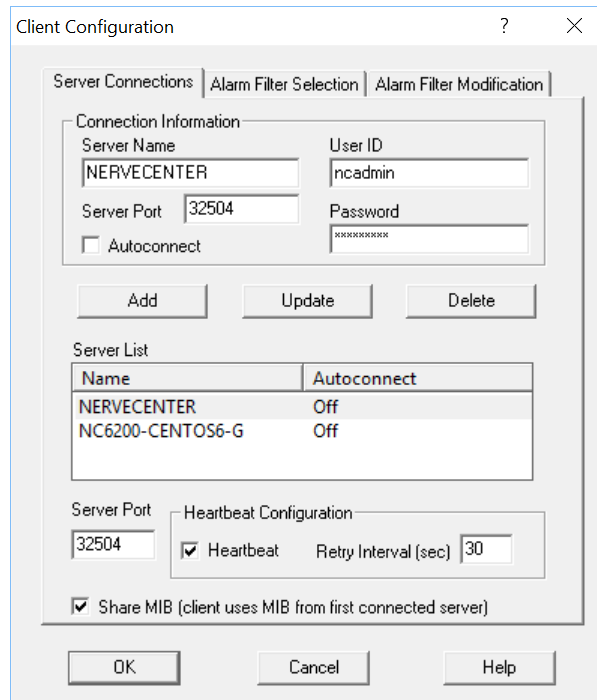
- A list of subnets
- A list of severities
- A list of property groups

For information on how to build an alarm-instance filter based on severities and property groups, see the respective section listed below:

- [Filtering Alarms by Severity on page 70](#)
- [Filtering Alarms by Property Groups on page 74](#)

TO CREATE AN ALARM FILTER BASED ON AN IP RANGE

1. Choose **Configuration** from the **Client** menu.
The Client Configuration dialog is displayed.



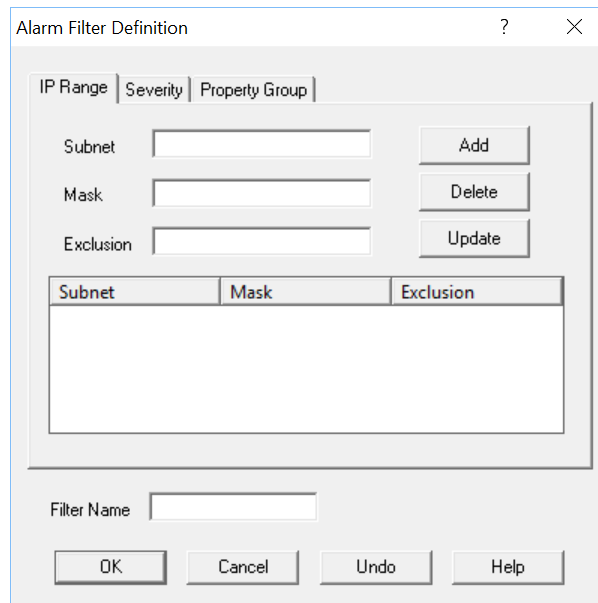
The screenshot shows the 'Client Configuration' dialog box with the 'Alarm Filter Modification' tab selected. The dialog is divided into several sections:

- Connection Information:** Fields for Server Name (NERVECENTER), User ID (ncadmin), Server Port (32504), and Password (masked with asterisks). There is an unchecked checkbox for 'Autoconnect'.
- Buttons:** 'Add', 'Update', and 'Delete' buttons are located below the Connection Information section.
- Server List:** A table with two columns: 'Name' and 'Autoconnect'. It lists two servers: 'NERVECENTER' (Autoconnect: Off) and 'NC6200-CENTOS6-G' (Autoconnect: Off).
- Heartbeat Configuration:** A section with a 'Server Port' field (32504) and a 'Heartbeat Configuration' area containing a checked 'Heartbeat' checkbox and a 'Retry Interval (sec)' field (30).
- Share MIB:** A checked checkbox labeled 'Share MIB (client uses MIB from first connected server)'.
- Buttons:** 'OK', 'Cancel', and 'Help' buttons are at the bottom of the dialog.

2. Select the **Alarm Filter Modification** tab.
The Alarm Filter Modification page is displayed.



3. Select the **New** button.
The Alarm Filter Definition dialog is displayed.



4. If you want to filter alarm instances based on the IP addresses of the alarm instances' nodes, perform the steps below for each subnet you want to be part of the filter. That is, you want to see information about instances whose nodes have IP addresses on these subnets.
 - a. Enter an IP address in the **Subnet** text field.
The IP address must consist of four octets separated by periods.
 - b. Enter a subnet mask in the **Mask** text field.
The subnet mask must consist of four octets separated by periods. Taken together with the subnet address, this mask defines the subnet whose nodes you're monitoring.
 - c. In the **Exclusion** text field, enter the last octet of the IP address of any node on the subnet that you're not monitoring.
You can enter multiple exclusions separated by commas. You can also enter a range of excluded nodes using a hyphen. For example, if you enter 24, 76-78 in the Exclusion field, the nodes whose addresses end in 24, 76, 77, and 78 will be excluded by the filter.
 - d. Select the **Add** button.
 - e. Repeat [Step a](#) to [Step d](#) to add other subnets to the alarm filter.
 5. Enter a name for your filter in the **Filter Name** field.
 6. Select the **OK** button.
The Alarm Filter Definition dialog is closed and you return to the Client Configuration dialog box.
-

You've now defined an alarm filter based on an IP range. Before the client will use the filter, however, you must associate the filter with a server. For instructions on how to create this association, see the section [Associating a Filter with a Server on page 77](#).

IP Subnet Filter Exclusion Rules

When you filter by subnet, you specify which subsets of nodes are managed by NerveCenter. Filtering does not apply to nodes that have been imported from a file or from another NerveCenter. For an example, see [IP Subnet Filter Examples on page 68](#).

You can exclude specific nodes that belong to the filter by entering an exclusion. To exclude one or more nodes, you must specify the full subnet and mask, and then enter the individual nodes you want excluded. Enter the part of the IP address that is not affected by the subnet's mask.

NerveCenter filters Class B and C networks.

- **In a Class C network**, the first three octets of the address specify the network and the last octet specifies the host. For example, in network 194.123.45.0, the 194.123.45 value pertains to the network. The remaining octet is used to identify nodes (up to 254) on the network, and you can exclude nodes by specifying ID values in this octet.
- **In a Class B network**, only the first two octets of the address specify the network. For example, in network 132.45.0.0, the 132.45 value pertains to the network. The remaining two octets are used to identify nodes, and you can exclude nodes by specifying ID values in these two octets.

Example

In the following example, the node whose IP address is 134.204.179.40 is excluded from the filter (the node is filtered out and, therefore, is not managed by NerveCenter).

```
134.204.179.0
255.255.255.0
40
```

Rules for Exclusions

- You can enter several nodes separated by a comma. NerveCenter accepts comma-separated values with or without spaces following the commas. You can enter the node values in any order.

The following three examples (each on a separate line) illustrate valid exclusions:

```
7, 8, 9, 15
7, 8, 9, 15
8, 7, 9, 15
```

- You can enter a range of values using a hyphen.

For example, you can enter as an exclusion range: **40-60**

You can also enter the range in inverse order: **60-40**

- You can include multiple entries for the same subnet if you have values or ranges that are not incremental.

- For example, you can enter as a filter:

```
134.204.179.0
255.255.255.0
7, 8, 9
134.204.179.0
255.255.255.0
40-60
134.204.179.0
255.255.255.0
70-90
```

- You can combine ranges, for example:

```
134.204.179.0
255.255.255.0
40-60, 70-90
```

- You can also combine formats, for example:

```
134.204.179.0
255.255.255.0
7-9, 31, 33, 40-60
```

IP Subnet Filter Examples

The following examples can help you understand how to filter nodes for Class B and C networks.

Class C Network

The following subnet filters are for two sample nodes:

- Sample node #1 with IP address: 197.204.179.25
- Sample node #2 with two IP addresses:
 - 134.204.179.40
 - 197.204.179.7

The filter values in [Table 9](#) have the following effects on the sample nodes:

Table 9: Class C Network Examples

Subnet Mask Exclusion	Results of Filter
134.204.179.0 255.255.255.0	This filter does not contain any exclusions. Node #1 is not on this subnet and is not included in the filter or managed by NerveCenter. Node #2 is included in the filter because it's on the subnet.
134.204.179.0 255.255.255.0 25,40	Node #1 is not on this subnet and is not included in the filter. Node #2 is listed as an exclusion and is not included in the filter.
197.204.179.0 255.255.255.0 7-20	Node #1 is included. Node #2 is not included because it's listed in the exclusion range.
197.204.179.0 255.255.255.0 7-20 134.204.179.0 255.255.255.0 40	Node #1 is included in the first subnet. Node #2 is not included because it's listed as an exclusion on both subnets.
197.204.179.0 255.255.255.0 25,40	Node #1 is not included because it's listed as an exclusion. Node #2 is included.

Class B Filters

The following subnet filters are for two sample nodes:

- Sample node #1 with IP address: 132.45.160.10
- Sample node #2 with IP address: 132.45.174.10

The mask you use for this filter is 255.255.0.0.

Table 10: Class B Filter Examples (Set One)

Subnet Mask Exclusion	Results of Filter
132.45.0.0 255.255.0.0	Both nodes are included in the filter and managed by NerveCenter.
132.45.0.0 255.255.0.0 174.10	Node #1 is included in the filter. Node #2 is excluded from the filter. The filter includes all nodes except 132.45.174.10.
132.45.0.0 255.255.0.0 160.10- 174.5	Node #1 is listed in the exclusion range and is excluded from the filter. Node #2 is included in the filter.
132.45.0.0 255.255.0.0 10	Both nodes are excluded from the filter and, therefore, neither node is managed by NerveCenter. The filter includes all nodes except 132.45.xxx.10, where xxx can be any value greater than 1 and less than 255.

If you use a subnet mask of 255.255.240.0, you would get different results.

- Sample node #1 with IP address: 132.45.160.10
- Sample node #2 with IP address: 132.45.174.10

You must first apply the filter before determining the node's ID. The filter values in the table below have the following effects:

Table 11: Class B Filter Examples (Set Two)

Subnet Mask Exclusion	Results of Filter
132.45.160.0 255.255.240.0 174.10	The node is not included in the filter. The filter includes all nodes except 132.45.174.10.
132.45.160.0 255.255.240.0 10	Neither node is included in the filter. The filter includes all nodes except those ending in .10. The third octet of an excluded node can be 174 or any value between 160 and 174.

Filtering Alarms by Severity

When you filter alarms by severity, you are specifying that you only want to display alarm instances in the NerveCenter Client from particular nodes identified by the severity of the alarm instance's state.

Although you can create a filter simply based on severity, a single filter can contain any combination of:

- A list of subnets
- A list of severities
- A list of property groups

For information on how to build an alarm-instance filter based on IP range and property groups, see the respective section listed below:

- [Filtering Alarms by IP Range on page 63](#)
- [Filtering Alarms by Property Groups on page 74](#)

TO CREATE AN ALARM FILTER BASED ON SEVERITY

1. Choose **Configuration** from the **Client** menu.
The Client Configuration dialog is displayed.

Client Configuration

Server Connections | Alarm Filter Selection | Alarm Filter Modification

Connection Information

Server Name: NERVECENTER User ID:

Server Port: Password:

Autoconnect

Add Update Delete

Server List

Name	Autoconnect
NERVECENTER	Off

Server Port: 32504

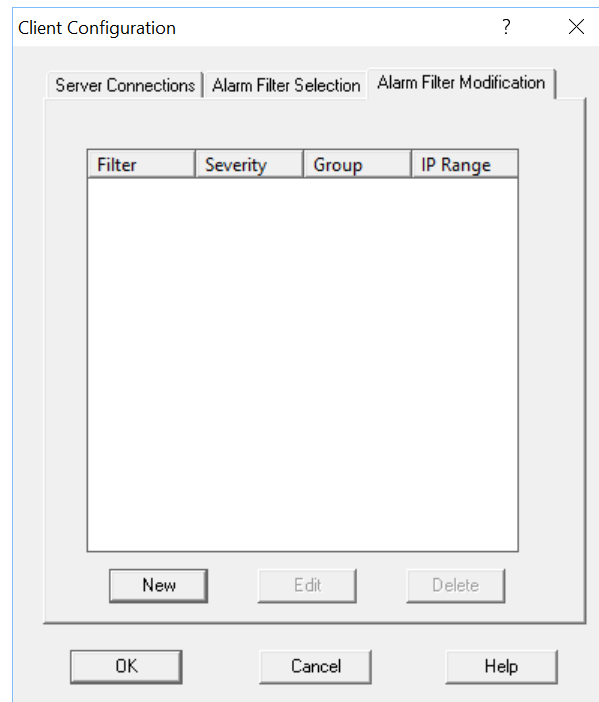
Heartbeat Configuration

Heartbeat Retry Interval (sec): 30

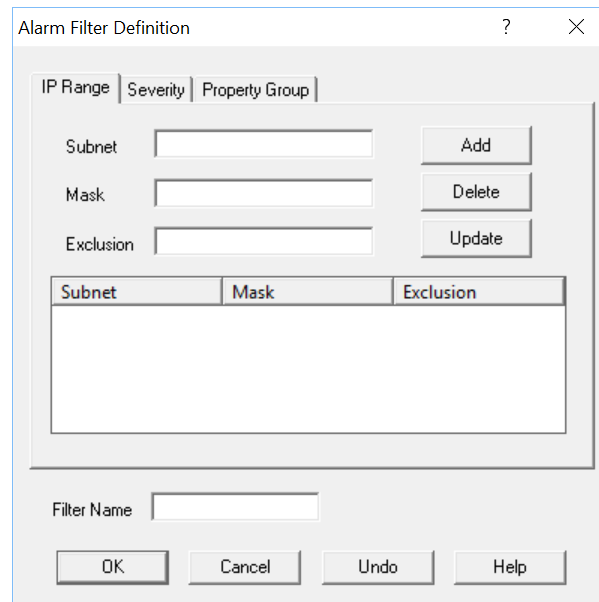
Share MIB (client uses MIB from first connected server)

OK Cancel Help

2. Select the **Alarm Filter Modification** tab.
The Alarm Filter Modification page is displayed.

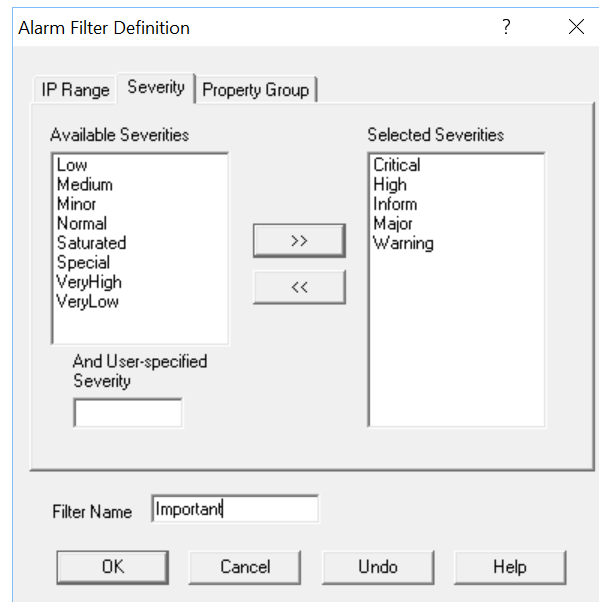


3. Select the **New** button.
The Alarm Filter Definition dialog is displayed.



This is the dialog you use to define your filter.

- Select the **Severity** tab.
The Severity tab is displayed.



- In the **Available Severities** list, for each severity you want to use in your filter, select the severity and then select the >> button. That is, you want to see information about alarm instances whose states have these severities.

The severities in this list box are the union of the severities defined by all of the servers to which you're connected. You can also add a user-defined severity to the list of severities to filter by entering it in the **And User-specified Severity** text box, and then clicking >>.

The name of the severity is moved to the **Selected Severities** list. Information about alarm instances with this severity will be displayed in the alarm summary views.

To remove a severity from the **Selected Severities** list, select the severity and then click <<.

- Enter a name for your filter in the **Filter Name** field.
- Select the **OK** button.
You return to the Client Configuration dialog box.

You've now defined an alarm filter based on severity. Before the client will use the filter, however, you must associate the filter with a server. For instructions on how to create this association, see the section [Associating a Filter with a Server on page 77](#).

Filtering Alarms by Property Groups

When you filter alarms by property groups, you are displaying alarm instances in the NerveCenter Client from particular nodes belonging to one or more property groups. While you can create a filter based on membership within a property group, a single filter can contain any combination of subnets, severities, or property groups.

For more on building an alarm-instance filter based on an IP range and severities, see the respective section listed below:

- [Filtering Alarms by IP Range on page 63](#)
- [Filtering Alarms by Severity on page 70](#)

TO CREATE AN ALARM FILTER BASED ON PROPERTY GROUPS

1. Choose **Configuration** from the **Client** menu.
The Client Configuration dialog is displayed.

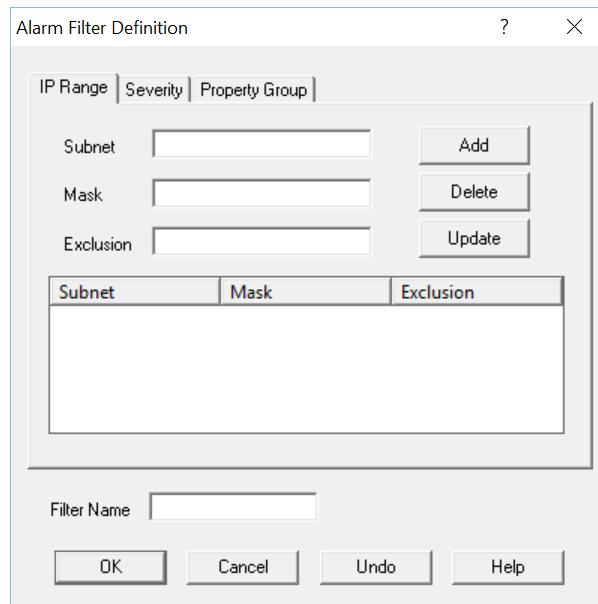
The screenshot shows the 'Client Configuration' dialog box with the 'Alarm Filter Selection' tab selected. The dialog is divided into several sections:

- Server Connections:** Contains three tabs: 'Server Connections', 'Alarm Filter Selection', and 'Alarm Filter Modification'.
- Connection Information:** A form with fields for 'Server Name' (containing 'NERVECENTER'), 'User ID', 'Server Port', and 'Password'. There is also an 'Autoconnect' checkbox.
- Buttons:** 'Add', 'Update', and 'Delete' buttons are located below the connection information.
- Server List:** A table with two columns: 'Name' and 'Autoconnect'. It contains one entry: 'NERVECENTER' with 'Off' in the 'Autoconnect' column.
- Server Port:** A text box containing '32504'.
- Heartbeat Configuration:** A section with a checked 'Heartbeat' checkbox and a 'Retry Interval (sec)' text box containing '30'.
- Share MIB:** A checkbox labeled 'Share MIB (client uses MIB from first connected server)'.
- Bottom Buttons:** 'OK', 'Cancel', and 'Help' buttons.

2. Select the **Alarm Filter Modification** tab.
The Alarm Filter Modification tab is displayed.



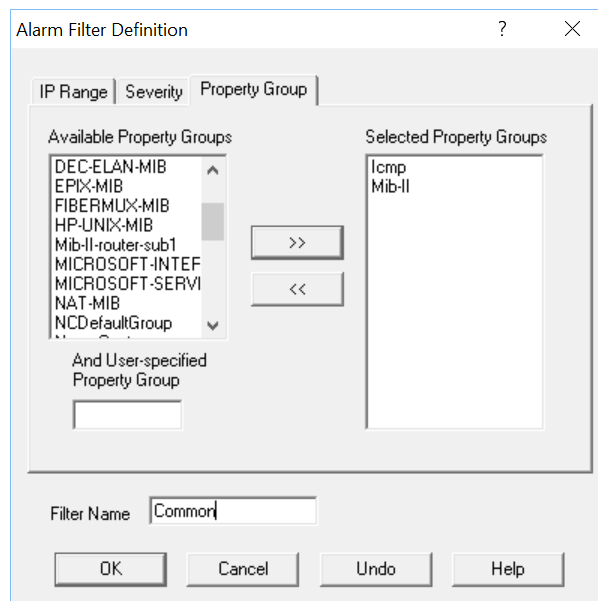
3. Select the **New** button.
The Alarm Filter Definition dialog is displayed.



The screenshot shows the "Alarm Filter Definition" dialog box with the "IP Range" tab selected. The dialog has three tabs: "IP Range", "Severity", and "Property Group". Under the "IP Range" tab, there are three input fields: "Subnet", "Mask", and "Exclusion". To the right of each field is a button: "Add" for Subnet, "Delete" for Mask, and "Update" for Exclusion. Below these fields is a table with three columns: "Subnet", "Mask", and "Exclusion". The table is currently empty. At the bottom of the dialog, there is a "Filter Name" input field and four buttons: "OK", "Cancel", "Undo", and "Help".

This is the dialog you use to define your filter.

4. Select the **Property Group** tab.
The Property Group tab is displayed.



The screenshot shows the "Alarm Filter Definition" dialog box with the "Property Group" tab selected. The dialog has three tabs: "IP Range", "Severity", and "Property Group". Under the "Property Group" tab, there are two lists: "Available Property Groups" and "Selected Property Groups". The "Available Property Groups" list contains: DEC-ELAN-MIB, EPIX-MIB, FIBERMUX-MIB, HP-UNIX-MIB, Mib-II-router-sub1, MICROSOFT-INTEF, MICROSOFT-SERVI, NAT-MIB, and NCDdefaultGroup. Below this list is a text input field labeled "And User-specified Property Group". Between the two lists are two buttons: ">>" and "<<". The "Selected Property Groups" list contains: Icmp and Mib-II. At the bottom of the dialog, there is a "Filter Name" input field containing the text "Common" and four buttons: "OK", "Cancel", "Undo", and "Help".

5. In the **Available Property Groups** list, for each property group of each alarm instance's node, perform the steps below for each property group you want to be part of the filter. That is, you want to see information about instances whose nodes belong to these property groups.

The property groups in this list box are the union of the property groups defined by all of the servers to which you're connected.

The property group is moved to the **Selected Property Groups** list. Information about alarm instances with this property will be displayed in the alarm summary views. Optionally, you can also add a user-defined property group to the list of properties to filter by entering a property group in the **And User-specified Property Group** text box, and then click **>>**.

To remove a property group from the **Selected Properties** list, select it and then click **<<**.

6. Enter a name for your filter in the **Filter Name** field.
7. Select the **OK** button.

You return to the Client Configuration dialog box.

You've now defined an alarm filter based on property groups. Before the client will use the filter, however, you must associate the filter with a server. For instructions on how to create this association, see the section [Associating a Filter with a Server](#) below.

Associating a Filter with a Server

When you define an alarm filter, that filter is not used to filter alarm instances from all connected servers. It is only used to filter alarm instances from a server with which you have explicitly associated it.

TO ASSOCIATE AN ALARM FILTER WITH A NERVECENTER SERVER

1. Choose **Configuration** from the **Client** menu.

The Client Configuration dialog is displayed.

Client Configuration

Server Connections | Alarm Filter Selection | Alarm Filter Modification

Connection Information

Server Name: NERVECENTER User ID:

Server Port: Password:

Autoconnect

Add Update Delete

Server List

Name	Autoconnect
NERVECENTER	Off

Server Port: 32504 Heartbeat Configuration: Heartbeat Retry Interval (sec): 30

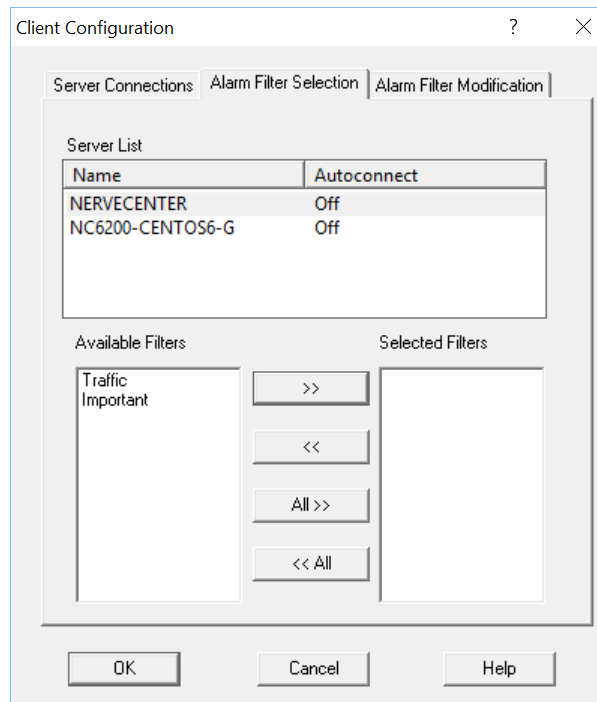
Share MIB (client uses MIB from first connected server)

OK Cancel Help

2. Select a server from the list of servers at the bottom of the dialog.

The name of the server appears in the **Server Name** text field in the Connection Information group box. This is the server with which you will associate your alarm filter.

3. Select the **Alarm Filter Selection** tab.
The Alarm Filter Selection page is displayed.



4. Select a filter from the **Available Filters** list.
This is the filter you want to associate with the server you selected in [Step 2](#).
5. Select the >> button to move the filter from the **Available Filters** list to the **Selected Filters** list.
To remove a filter from the **Selected Filters** list, select the filter and then select the << button.
6. Select the **OK** button at the bottom of the dialog.

Rules for Associating Filters with Alarms

When deciding whether to apply multiple filters to your alarms, you should keep in mind the following general rules:

- Multiple filters are ORed together
- Multiple conditions in a single filter are ANDed together

Multiple Filters are ORed Together

When you select more than one filter for a server, each filter is independent of the other filters. Their behavior is equivalent to a logical OR operation.

For example, say you associate two filters with a NerveCenter Server. The two filters are defined as follows:

- Filter #1 is configured to display only those alarms that have a severity level of Critical.
- Filter #2 is configured to display only those alarms coming from the network 132.168.196.0.

When both filters are applied to a server, you see the following alarms:

- Alarms with a Critical severity level from all existing networks defined for the server.
- From the network 132.168.196.0, you see all alarms regardless of severity.

Multiple Conditions in a Single Filter are ANDed Together

If, instead of the above view, you want to limit your alarms to Critical instances coming from the network 132.168.196.0, you need to create one filter with both of those conditions. You would create one filter that:

- Specifies a severity level of Critical, and
- Specifies an IP range of 132.168.196.0.

With this filter applied to the server, you see only those alarms that have a Critical severity level *and* that come from network 132.168.196.0. One filter with multiple conditions is equivalent to a logical AND operation; each condition is ANDed with the other conditions for optimum filtering.

Specifying Heartbeat Messaging

The NerveCenter Client sends a message called a *heartbeat* to each connected NerveCenter Server on a standard interval. This messaging ensures the reliability of communications between the server and client. If a server fails to respond after three consecutive heartbeat messages from the client, a message box is displayed on the client console to alert the operator of the server's heartbeat failure. (In such cases, you should check with your network administrator to obtain the status of that particular NerveCenter Server.)

You can set the interval at which the NerveCenter Client sends a heartbeat to the NerveCenter Server (30 seconds by default). You can also choose to deactivate heartbeat messaging.

See the following sections for more information:

- [Modifying the Heartbeat Message Interval on the facing page](#)
- [Deactivating Heartbeat Messaging on page 82](#)

Modifying the Heartbeat Message Interval

You can change the interval NerveCenter Client uses to send heartbeat messages to verify its connection with your NerveCenter Servers.

TO MODIFY THE HEARTBEAT MESSAGE INTERVAL

1. Choose **Configuration** from the **Client** menu.
The Client Configuration dialog is displayed.

Client Configuration

Server Connections | Alarm Filter Selection | Alarm Filter Modification

Connection Information

Server Name: NERVECENTER User ID:

Server Port: Password:

Autoconnect

Add Update Delete

Server List

Name	Autoconnect
NERVECENTER	Off

Server Port: 32504

Heartbeat Configuration

Heartbeat Retry Interval (sec): 30

Share MIB (client uses MIB from first connected server)

OK Cancel Help

2. In the **Heartbeat Configuration** panel, make sure the **Heartbeat** checkbox is checked. If it's not checked, heartbeat messaging is turned off.
3. In the **Retry Interval** field, enter the number of seconds you want NerveCenter Client to wait between heartbeat messages. The default is 30 seconds. (The number of retries is three.)

Note: When you modify heartbeat messaging, it applies to all NerveCenter Servers to which this client connects.

4. Select the **OK** button.

Deactivating Heartbeat Messaging

The NerveCenter Client sends heartbeat messages on an interval that you specify (or by default, every 30 seconds) to verify its connection with your NerveCenter Servers. If you choose, you can deactivate (or activate) heartbeat messages going to and from *all* your connected servers.

TO DEACTIVATE HEARTBEAT MESSAGES

1. Choose **Configuration** from the **Client** menu.
The Client Configuration dialog is displayed.

2. In the Heartbeat Configuration panel, uncheck the **Heartbeat** checkbox.

Note: If there is no check mark in this checkbox, heartbeat messaging has already been deactivated for NerveCenter Client. When you activate or deactivate heartbeat messaging, it applies to all NerveCenter Servers to which this client connects.

3. Select the **OK** button.

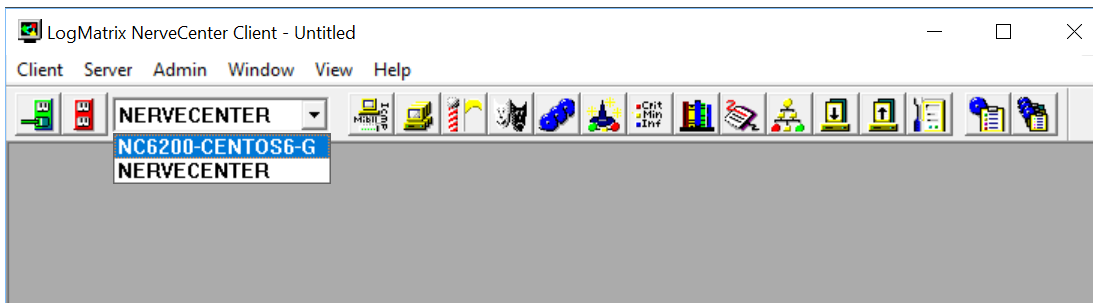
Heartbeat deactivation takes effect the next time you connect NerveCenter Client to one or more of your NerveCenter Servers.

Disconnecting from a Server

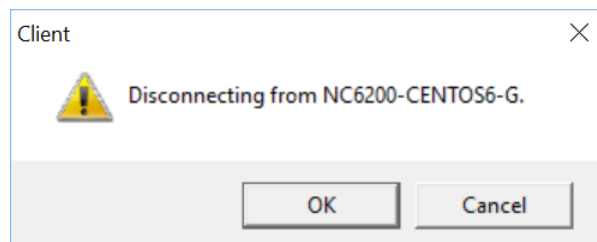
When you exit the client, all connections to NerveCenter servers are broken. However, you may also want to disconnect the client from a server without stopping the client.

TO DISCONNECT THE CLIENT FROM A SERVER

1. From the server drop-down list on the client's button bar, select the server with which you want to break the connection.



2. From the client's **Server** menu, choose **Disconnect**.
You see a pop-up window that asks you to confirm that you want to disconnect from the selected server.



3. Select the **OK** button.

Discovering and Defining Nodes

Before NerveCenter can manage a set of devices, a set of node definitions must reside in the NerveCenter database. There are two ways to enter these definitions into the NerveCenter database:

- By using a discovery mechanism. Both network management platforms and NerveCenter itself have the ability to explore a network and discover what devices are on the network. NerveCenter can use the information gleaned during this discovery process to create a set of node definitions.
- By defining the nodes manually using the NerveCenter GUI.

Generally, if you're managing a network of any size, you'll use a discovery mechanism to gather information about the devices on your networks. Defining nodes manually is appropriate only if you have a very small network or if you want to add to your database some nodes that were not found during the discovery process (perhaps because they were on a subnet that the discovery program did not explore).

Discovering Nodes

Generally, you add node definitions to the NerveCenter database using a discovery program. The two most common scenarios are listed below:

- You are using NerveCenter with a network management platform and you use the platform's discovery mechanism to explore the network and write node definitions to the platform's database. You then define the machine on which the platform is running as NerveCenter's *node source*. This action causes NerveCenter to copy the node definitions in the platform's database to its own database. The node information in NerveCenter's database is updated whenever the node information in the platform's database changes, for example, if a node is added to or deleted from the platform's database or if a node's attributes are changed.
- You are using NerveCenter in standalone mode, and you use NerveCenter's IPSweep behavior model to explore the network and write node definitions to NerveCenter's database.

There are also other, less common, scenarios. For example, you may be using NerveCenter with a network management platform, but NerveCenter may be set up at a remote site and the platform may be running at a central site. In this case, it may make sense to have NerveCenter discover the remote network and forward the node information it gathers to the platform. NerveCenter can then retrieve node definitions from the platform as in the first case mentioned above.

In any of these situations, you may only want information about nodes on particular subnets. This type of filtering is easy to do with NerveCenter; however, it must be set up from the NerveCenter Administrator. For information on how to perform this task, see [Filtering Nodes](#) in Managing NerveCenter

For more detailed information about discovering nodes, see [Using IPSweep Behavior Model on the facing page](#)

Using IPSweep Behavior Model

NerveCenter can be configured to discover nodes and add them to its database. With discovery enabled, if the NerveCenter database does not already have a node matching the source of an SNMP trap or NerveCenter Inform, it adds the node to the database. If NerveCenter Administrator is configured to auto-classify, NerveCenter also attempts to classify the node's SNMP version.

You typically use NerveCenter to discover nodes when NerveCenter is not integrated with a network management platform.

For times when you want NerveCenter to discover the devices on a network, NerveCenter includes the IPSweep behavior model. To use this behavior model, you—or for the first step, an administrator—must perform the following tasks:

1. Someone must specify the following information:
 - Which subnets the IPSweep behavior model should explore, and any nodes on those subnets that the model should ignore
 - Whether node information should be sent to NerveCenter or to a network management platform
 - Whether the IPSweep alarm should be started automatically when the NerveCenter Client is started.

This information can be specified either when NerveCenter is installed or later via the NerveCenter Administrator. For installation details, see [Installing NerveCenter](#), and for information about using the NerveCenter Administrator, see [Populating Using the IPSweep Behavior Model](#) in [Managing NerveCenter](#).

2. You must make minor changes to the predefined NerveCenter alarm: IPSweep.
3. You must enable the IPSweep alarm.

Once the IPSweep behavior model becomes operational, it finds devices on the subnets you've specified and, for each node, send a trap to the NerveCenter server or the network management platform. If the trap is sent to NerveCenter, the server creates a node definition and places it in the NerveCenter database. If the trap is sent to the platform, the platform writes information about the node to its database, and then that information becomes available to NerveCenter.

Both the customization and enabling of the IPSweep alarm is handled from the NerveCenter Client. For instructions on how to modify and enable these alarms, refer to the following sections:

- [Modifying the IPSweep Alarm on the next page](#)
- [Enabling the IPSweep Alarm on page 90](#)

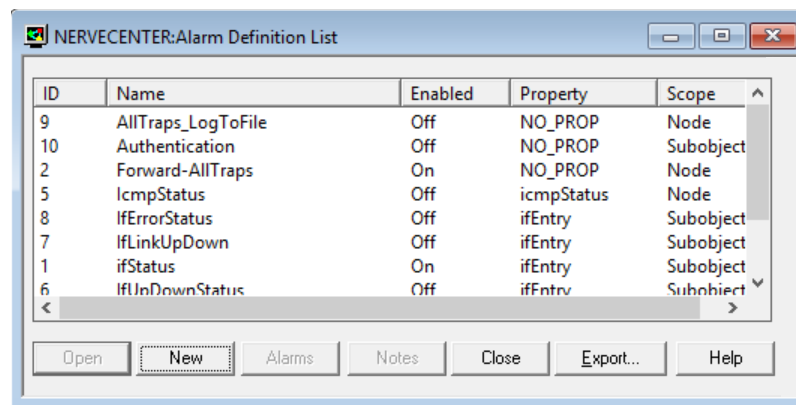
Modifying the IPSweep Alarm

The IPSweep alarm actually executes the program, `ipsweep`, that discovers devices on your network. If NerveCenter was installed in the default directory, this alarm will work correctly without modification. However, if the product was installed in a non-default directory, you must change the Command action associated with one of the alarm's transitions so that the path to `ipsweep` is correct. You may also want to change the delay between executions of the `ipsweep` program. The instructions below explain how to change both the delay and the path to the `ipsweep` program.

TO MODIFY THE IPSWEEP ALARM

1. From the client's **Admin** menu, choose **Alarm Definition List**.

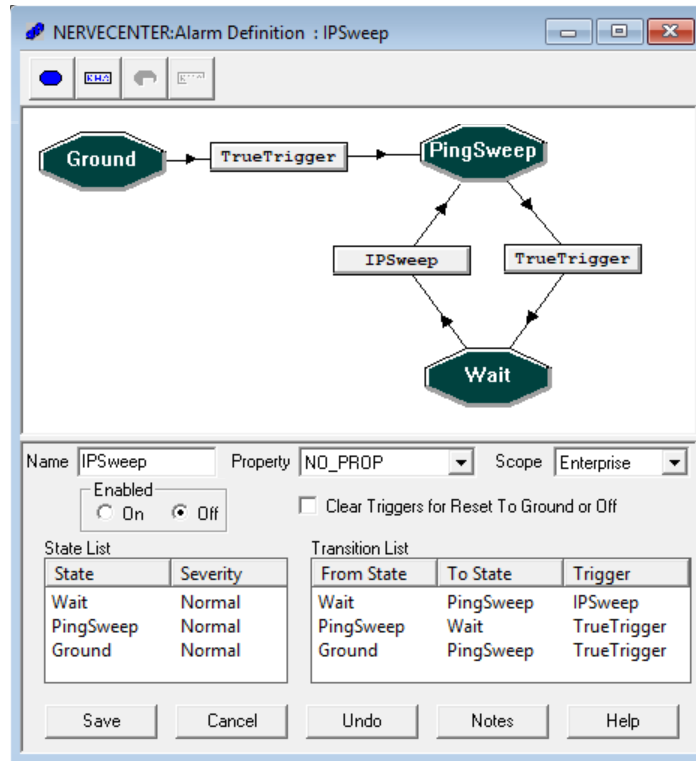
The Alarm Definition List window is displayed



2. Select the IPSweep alarm from the list.

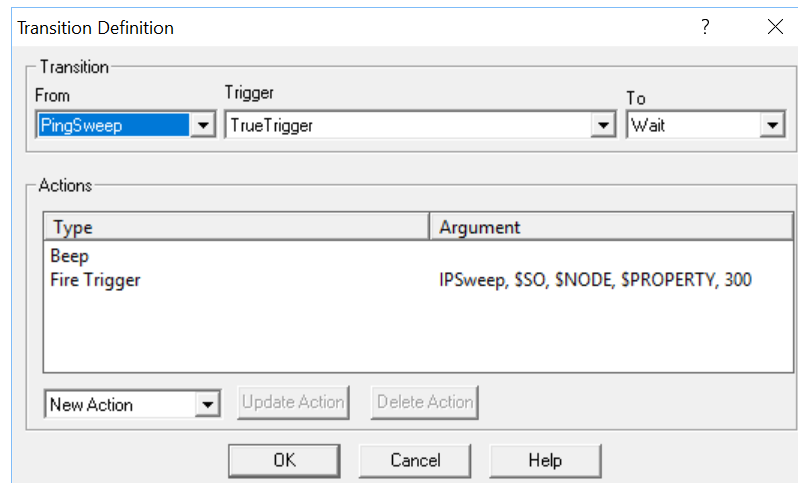
The Open button is enabled.

3. Select the **Open** button.
The definition of the IPSweep alarm is displayed in the Alarm Definition window.

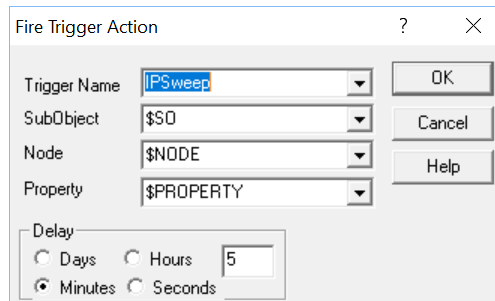


4. If the alarm is enabled, set its Enabled status to **Off**.
The alarm may be turned on even if it was never explicitly enabled. This is possible because NerveCenter server can be configured to enable this alarm on startup.

- Double-click the transition from the PingSweep state to the Wait state.
The Transition Definition dialog is displayed.

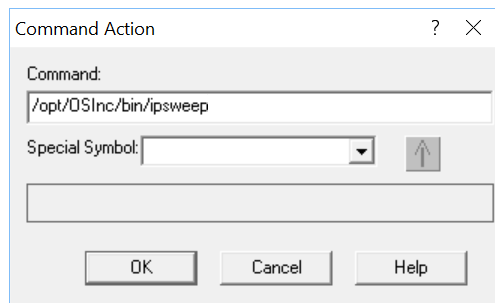


- Double-click the Fire Trigger action.
The Fire Trigger Action dialog is displayed.



- Change the delay for the Fire Trigger action from 5 minutes to the length of time you want to wait between invocations of the ipsweep program.
A short delay will generate more network traffic, while a long delay will mean a longer wait for new devices to be discovered.
- Select the **OK** button in the Fire Trigger Action window.
- Select the **OK** button in the Transition Definition window.

10. Double-click the IPSweep transition.
The Transition Definition window is displayed.
11. Double-click the Process Command action in the Transition Definition window.
The Command Action dialog is displayed.



12. Edit the **Command** text field so that it contains the correct path to the ipsweep program.
 13. Select the **OK** button in the Command Action window.
 14. Select the **OK** button in the Transition Definition window.
 15. Select the **Save** button in the Alarm Definition window.
-

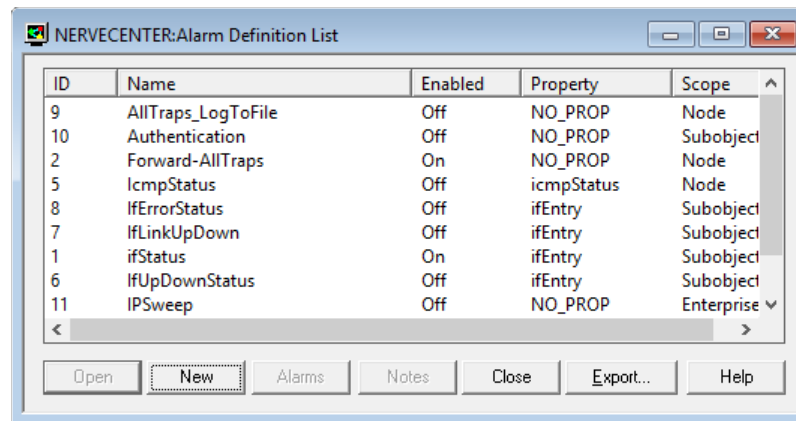
Enabling the IPSweep Alarm

Once you've modified the IPSweep alarm, you must enable the alarm for the IPSweep behavior model to become functional.

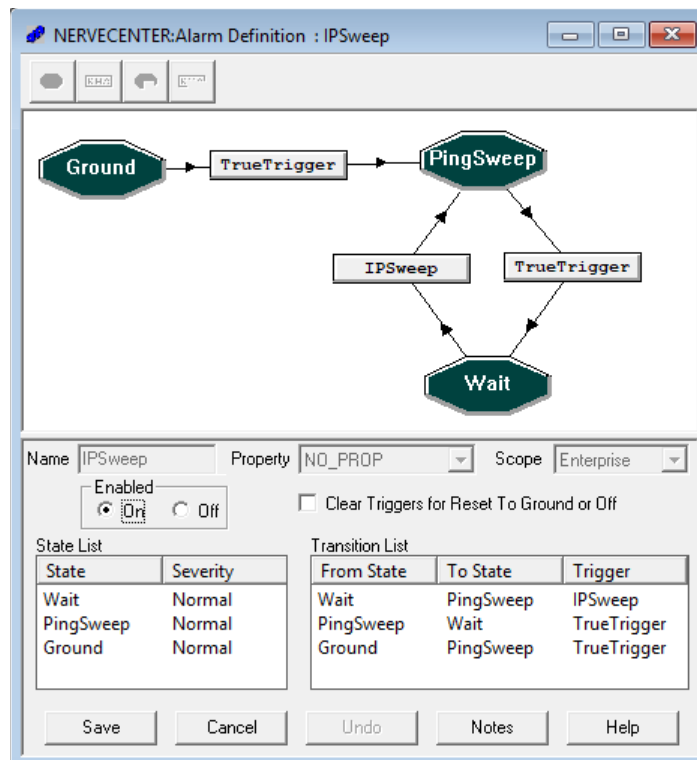
TO ENABLE THE IPSWEEP ALARM

1. *For each alarm*, perform this step and the following steps. From the client's **Admin** menu, select **Alarm Definition List**.

The Alarm Definition List window is displayed.



2. Highlight the name of the alarm you want to enable.
The Open button is enabled.
3. Select the **Open** button.
The alarm's definition is displayed in the Alarm Definition window.



4. Select the **On** radio button in the Enabled frame.
5. Select the **Save** button at the bottom of the window.

Note: You can also enable an alarm by selecting it in the Alarm Definition list, pressing the right mouse button while your cursor is positioned over the highlighted alarm, and selecting On from the pop-up menu.

Defining Nodes Manually

There are three situations in which you should define nodes manually using the NerveCenter Client.

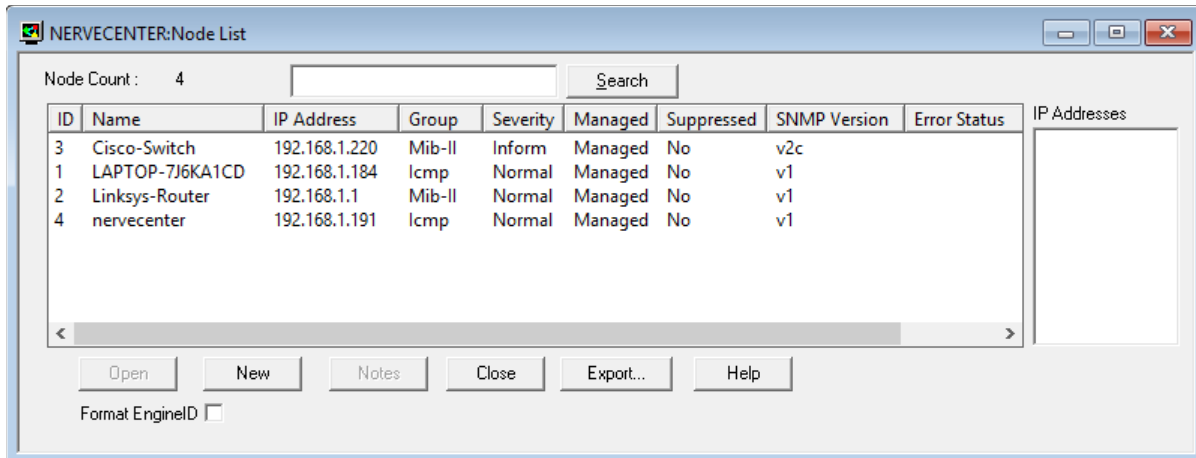
- You are managing a very small network, and it is easier to define the nodes in the network manually than to configure NerveCenter's IPSweep behavior model.
- You've discovered most of your nodes using either your network management platform's or NerveCenter's discovery mechanism, but you need to add to your database a few nodes on a subnet that wasn't explored during the discovery process.
- You are managing a network with IPv6 support.

In any case, you can define your nodes using the Node Definition window in the client.

TO DEFINE A NODE MANUALLY

1. From the client's **Admin** menu, select **Node List**.

The Node List window is displayed.



- In the Node List window, select the **New** button.
The Node Definition window appears.

The screenshot shows the 'NERVECENTER:Node Definition' dialog box. It features a menu bar with 'Node', 'Alarms', 'Alarm Instances', 'MIB Query', 'Parents', 'SNMP', 'ICMP', and 'Tree'. The main area includes a 'Node ID' field, a 'Name' text field, a 'Property Group' dropdown menu (set to 'NCDefaultGroup'), and an 'IP Address' text field. To the right are checkboxes for 'Managed', 'Autodelete' (checked), 'Suppressed', and 'Platform Node'. Below the IP Address field are buttons for 'Add', 'Update', 'Delete', 'IP Lookup', and 'StopLookup'. An 'IP Address List' box is located below these buttons. At the bottom of the dialog are buttons for 'Save', 'Cancel', 'Undo', 'Notes', and 'Help'.

- In the **Name** text field, type the name of the workstation or network device that the node object represents. The name can be a fully qualified hostname or an IP address.

Note: The maximum length for node names is 255 characters.

- Select the node's property group from the **Group** list box.
The **Group** list box contains a list of all the valid property group names defined in the NerveCenter database.
- In the **Port** text field, type the number of the port on the node to which NerveCenter should send messages.
SNMP agents use port 161 to receive SNMP messages.

6. In the **IP Address** text field, type the node's IP address. Then select the **Add** button to add the address to the IP Address List. If the node is multihomed, you can add the node's other addresses to the list in the same manner.

If you need to delete an address from the address list, highlight that address, and then select the **Delete** button.

If you have a properly configured DNS or NIS server, you can also use IP Lookup to find the IP Address(es) for the node. See [Using IP Lookup on the facing page](#) for more details.

7. Check the **Managed** checkbox if you want NerveCenter to manage the node.
You can leave **Managed** unchecked if you do not want the node to be affected by any NerveCenter behavior models.
8. Check the **Auto Delete** checkbox if you want the node to be deleted if it is not in your network management platform's (NMP's) node database.
The setting of this property is meaningful only if you are using an NMP as your node source. If you're using an NMP as a node source and you check the Auto Delete checkbox, the node you're defining will be deleted when the NerveCenter database is synchronized with the NMP's node database, *if the node you're defining is not found in the NMP's node database*. If you don't want the node to be deleted in this situation, don't check the Auto Delete checkbox.
9. The **Platform** checkbox is a read-only control.
When you define a node manually, **Platform** is read-only and is unchecked and indicates that the node you are defining was not discovered by a network management platform.
10. Check the **Suppressed** checkbox if you want the node to be in a suppressed state.
A suppressed node is not polled by any suppressible polls (a poll's default state). Only polls designed to monitor a device's responsive/unresponsive state are not suppressible.

Note: Normally, you do not check **Suppressed**. A node's suppressed attribute is usually set by an alarm action when the alarm detects that the node is not reachable.

11. By default, NerveCenter deems the SNMP version for a node to be version 1. If you want to manage the node using SNMP version 2c or 3, you must configure the appropriate SNMP settings in the SNMP tab. In the SNMP tab, you can also change the Read and Write community names for a node that's using SNMP version 1 or 2c.

For details, see [Configuring SNMP Settings for Nodes on page 99](#).

12. Select the **Save** button.

When adding nodes manually, you can also search for the IP Addresses available for the hostname of the node.

Note: IP Lookup functions only if you have a properly configured DNS or NIS server. Connectivity is not enough for IP Lookup to find the correct IP address.
IP Lookup is separate from the DNS Lookup used by NerveCenter when discovering nodes from traps. See [Managing Node Data](#) in Managing NerveCenter for more details about DNS Lookup.

USING IP LOOKUP

1. From the client's **Admin** menu, select **Node List**.
The Node List window is displayed.
2. In the Node List window, select the **New** button or select a Node and select **Open**.
The Node Definition window appears.

NERVECENTER:Node Definition

Node | Alarms | Alarm Instances | MIB Query | Parents | SNMP | ICMP | Triggers

Node ID: _____

Name: _____

Property Group: NCDefaultGroup

Managed

Autodelete

Suppressed

Platform Node

IP Address: _____

IP Address

Add Update Delete IP Lookup StopLookup

IP Address List

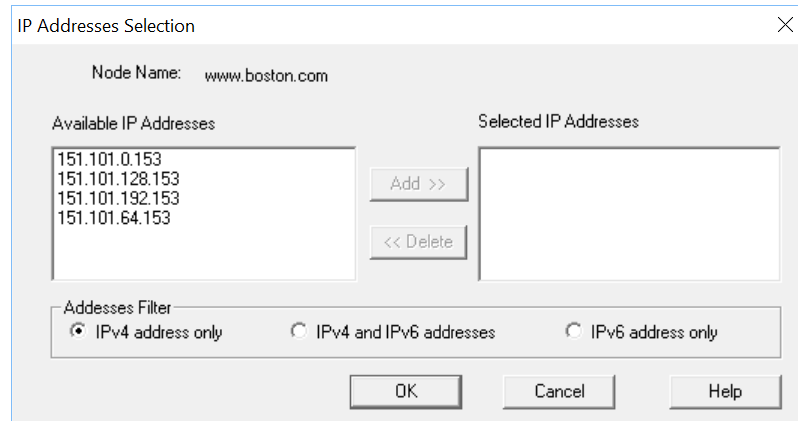
Save Cancel Undo Notes Help

3. In the **Name** text field, type the fully qualified name of the workstation or network device that the node object represents.

4. Select **IP Lookup**.

The IP Address Selection window opens.

If the IP Address Selection window does not open, you can click **StopLookup** to stop the the IP lookup and return control to the NerveCenter client.



If any node IP Addresses have been defined, they appear in the Selected IP Addresses field.

5. If you already defined IP addresses for the node, select them all and select **Delete** to move them to the Available IP Addresses field.
6. Select what type of IP Addresses you want to collect, IPv4, IPv6, or a combination of IPv4 and IPv6 addresses.

See [IPv6 and NerveCenter on the facing page](#) for more details about NerveCenter's IPv6 support.

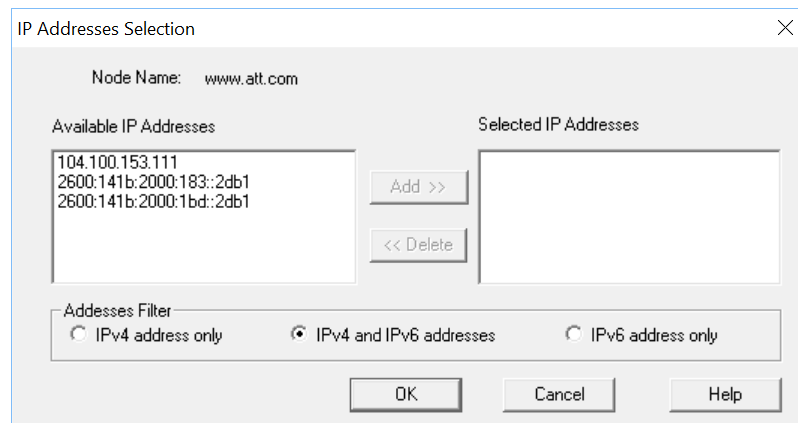
Note: If you have any invalid IP Addresses in the Available IP Addresses field—for example, if you defined IP Address manually in the Node Definition window—NerveCenter deletes them when you select an Address Filter.

7. Select the IP Addresses you want to assign to the Node from the Available IP Addresses field and click **Add** to move them to the Selected IP Addresses field.
8. Select **OK** to return to the Node Definition window.
9. Select **Save**.

IPv6 and NerveCenter

NerveCenter supports networks running IPv6 and IPv4 within the following guidelines:

- Your network must support the IPv6 protocol. If IPv6 devices reside on different network than the NerveCenter management node, your routers must support IPv6 routing.
- You must configure The SNMP agent on your IPv6 devices to support all SNMP requests on IPv6.
- You must install the NerveCenter server on a machine with IPv6 and IPv4 stack support.
- You must have a working DNS, NIS or other server properly configured to use name resolution for IPv6 addresses.
- You must install the NerveCenter server on a machine with IPv6 and IPv4 stack support.



While NerveCenter supports monitoring IPv6 networks, it has the following limitations:

- Communication between the NerveCenter server, the Administrator, and the Client does *not* support IPv6.
- Communication between NerveCenter and network management platforms does *not* support IPv6.
- The IPSweep alarm does *not* support IPv6.

Configuring SNMP Settings for Nodes

A node must have SNMP version information before NerveCenter can poll the node or process a trap from the node. If the node is using SNMPv3, the SNMP agent must be configured properly on the node. See [Using the SNMP Test Version Poll on page 126](#) for help testing communication with a node.

You can manually specify the correct SNMP version for the node or command NerveCenter to classify the node. If you specify the node as SNMPv3 or if the node is classified as SNMPv3, you can set the security level, user name, context and, if applicable, the authentication and privacy protocols used by NerveCenter to poll the node.

Manually Changing the SNMP Version Used to Manage a Node

NerveCenter must use different versions of SNMP to communicate with the different versions of SNMP agents. Most often, you will want NerveCenter to classify the SNMP version for nodes when they are added to your database. You can, however, manually change the version that NerveCenter uses for communicating with a particular node.

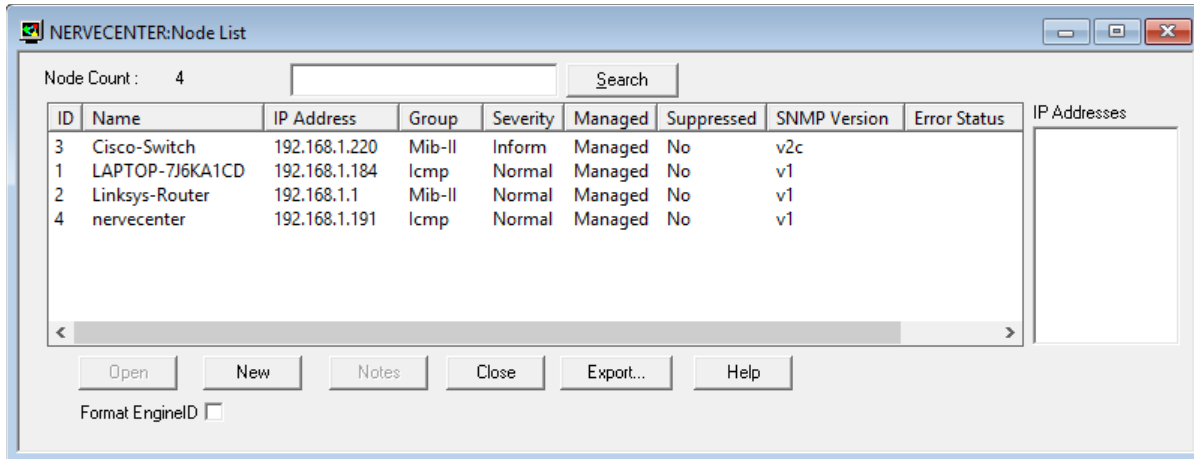
You might change the version, for example, if you are using SNMPv3 and it is not configured correctly at the agent. Instead of continuing to send SNMPv3 polls that may generate numerous alarms, you can temporarily change the node's SNMP version to v1 or v2c until you have a chance to reconfigure the v3 information at the agent. With this change, you can still poll the node for certain MIB variables defined in your behavior models and continue monitoring minimal MIB information for the node.

This feature also allows you to override the maximum version classification value configured in NerveCenter Administrator. For example, if maximum classification value is v2c, you can specify SNMPv3 for a particular node and run a test poll against that node.

TO CHANGE A NODE'S SNMP VERSION MANUALLY

1. From the client's **Admin** menu, select **Node List**.

The Node List window is displayed.



- In the Node List window, select **New** if defining a new node, or select the node and then **Open** to change an existing node.

The Node Definition window appears.

- Select the **SNMP** tab.

The screenshot shows the 'NERVECENTER:Node Definition : Cisco-WirelessVPNRouter' window with the 'SNMP' tab selected. The interface includes several sections for configuration:

- SNMPv2c**: A dropdown menu set to 'SNMPv2c', a 'Classify' button, a 'Port' field with '161', a '/udp' checkbox, and a 'Use GetBulk' checkbox which is checked.
- SNMP v1/v2c**: Fields for 'Read Community' and 'Write Community', both set to 'public'.
- SNMP v3**: A 'Select User' dropdown menu and a 'Security Level' dropdown menu.
- Local User**: A sub-section with fields for 'User Name', 'Authentication' (set to 'MD5'), 'Privacy' (set to 'DES'), and 'Context'. There are buttons for 'Set Auth Key' and 'Set Priv Key'.
- Engine ID**: A text field with 'Parse' and 'Fetch' buttons.
- Status**: A text field.

At the bottom of the window are buttons for 'Save', 'Cancel', 'Undo', 'Notes', and 'Help'.

- Select the node's **SNMP Version**.
 - If you select **Unknown** or an incorrect version, NerveCenter cannot poll the node or process traps from the node.
 - If you select v1 or v2, set the Read Community and Write Community values as appropriate for the node.
 - If you select v3, click the Select User list and select User #1, User #2, or Local User; and select the appropriate Security Level. If you select Local User, you can configure a node-specific username, authentication and privacy settings, and context as necessary.

5. Select the **Save** button.

Caution: NerveCenter performs no type of error check to validate the version you choose. However, you can manually confirm SNMPv3 communication with the node by clicking **Save** and then testing this setting with **Get** on the Query Node tab.

Note: To change the version of one or more nodes from the Node List window, right-click one or more nodes, select Version, and select the version you want for the nodes.

Changing the Security Level of an SNMPv3 Node

NerveCenter lets you set the security level you want for each managed node using SNMPv3. The security level of a node determines whether authentication or encryption services are used with communications between NerveCenter and the node.

SNMPv3 nodes can have one of the following security levels:

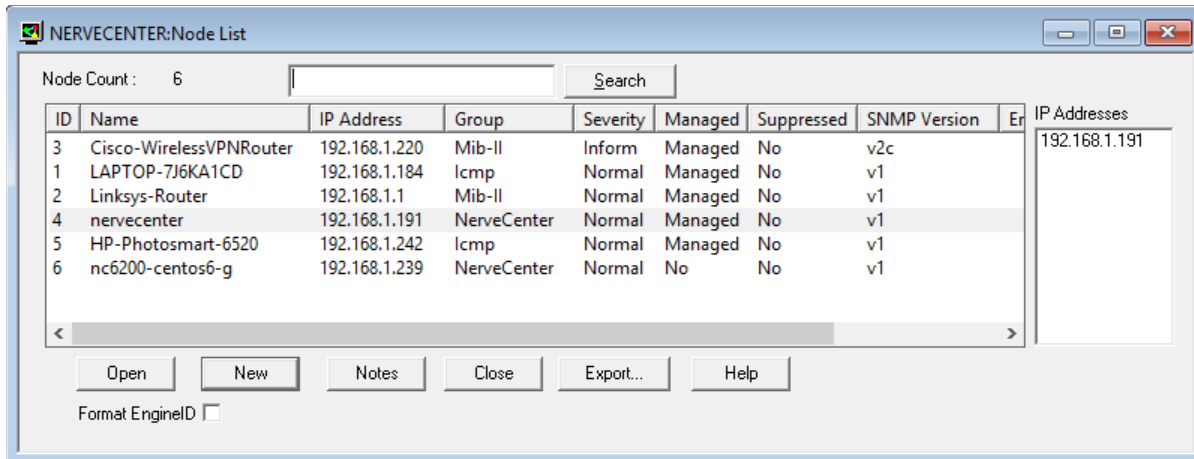
- **NoAuthNoPriv**—Neither message authentication nor encryption is used while communicating with the agent. No passwords are required.
- **AuthNoPriv**—Message authentication is used without encryption while communicating with the agent. An authentication protocol and password are required. The authentication password can be set to one of the two global passwords defined under User #1 or User #2, or can be set on a per-node basis.
- **AuthPriv**—Both authentication and encryption are used when communicating with the agent. Both the authentication and privacy protocols and passwords are required. These passwords can be set to one of the two global passwords defined under User #1 or User #2, or can be set on a per-node basis.

For more information on SNMPv3 security, see [NerveCenter Support for SNMPv3 Security on page 118](#). For details about passwords, see [NerveCenter Support for SNMPv3 Digest Keys and Passwords on page 119](#).

TO CHANGE AN SNMPV3 NODE'S SECURITY LEVEL

1. From the client's **Admin** menu, select **Node List**.

The Node List window is displayed.



- In the Node List window, select **New** if defining a new node, or select the node and then **Open** to change an existing node.
The Node Definition window appears.
- Select the **SNMP** tab.

The screenshot shows the 'NERVECENTER:Node Definition : Cisco-WirelessVPNRouter' window with the 'SNMP' tab selected. The window contains the following fields and controls:

- SNMPv3** dropdown menu and **Classify** button.
- Port** field set to 161, **/udp** checkbox, and **Use GetBulk** checkbox (checked).
- SNMP v1/v2c** section with **Read Community** and **Write Community** fields, both set to 'public'.
- SNMP v3** section with **Select User** dropdown (Local User) and **Security Level** dropdown (NoAuthNoPriv).
- Local User** section with **User Name** field (nervecenter), **Authentication** dropdown (MD5) with **Set Auth Key** button, **Privacy** dropdown (DES) with **Set Priv Key** button, and **Context** field.
- Engine ID** field with **Parse** and **Fetch** buttons.
- Status** field.
- Polling** section with **Use Defaults** checkbox (checked), **Retry Interval (seconds)** field, **Attempts** field, and **Timeout (seconds)** field.
- Bottom buttons: **Save**, **Cancel**, **Undo**, **Notes**, and **Help**.

- Select the **Security Level**.

If you are setting the security level for the Local User, you can configure the necessary authentication and privacy keys from this screen. If you are setting the security level for User #1 or User #2, those keys are configured from the SNMPv3 tab in the NerveCenter Administrator (see [Configuring SNMPv3 Security Settings](#) in Managing NerveCenter).

- Select the **Save** button.

Note: You can change node security levels from the Node List window. Right-click one or more nodes, select **Security Level**, and then select the level you want for the nodes.

Changing the Authentication Protocol for an SNMPv3 Node

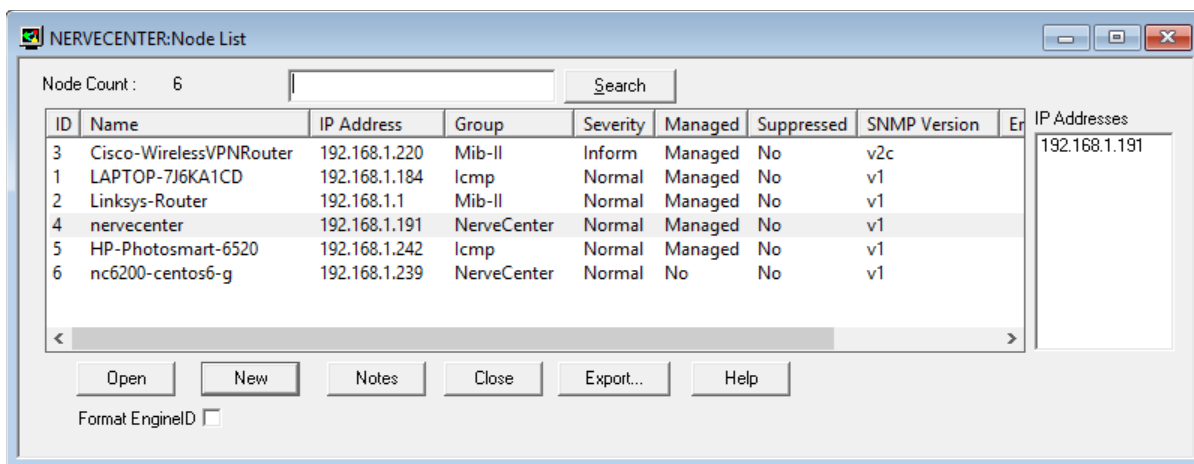
If you change the authentication protocol on an SNMPv3 agent, you must likewise change the protocol used by NerveCenter to manage that agent.

An authentication protocol must be specified when the node's security level is AuthNoPriv or AuthPriv. NerveCenter supports either HMAC-MD5-96 (MD5) or HMAC-SHA-96 (SHA-1) as authentication protocols. The default is MD5.

TO CHANGE THE AUTHENTICATION PROTOCOL USED TO MANAGE AN SNMPv3 NODE

1. From the client's **Admin** menu, select **Node List**.

The Node List window is displayed.



2. In the Node List window, select **New** if defining a new node, or select the node and then **Open** to change an existing node.

The Node Definition window appears.

3. Select the **SNMP** tab.

The screenshot shows the 'NERVECENTER:Node Definition : Cisco-WirelessVPNRouter' window with the 'SNMP' tab selected. The interface includes the following elements:

- SNMPv3** dropdown menu and a **Classify** button.
- Port** field set to 161, **/udp** checkbox, and **Use GetBulk** checkbox (checked).
- SNMP v1/v2c** section with **Read Community** and **Write Community** fields, both set to 'public'.
- SNMP v3** section with **Select User** dropdown (Local User), **Security Level** dropdown (AuthNoPriv), and a **Local User** sub-section containing:
 - User Name** field: nervecenter
 - Authentication** dropdown: SHA-1, with a **Set Auth Key** button.
 - Privacy** dropdown: AES-128, with a **Set Priv Key** button.
 - Context** field.
- Engine ID** field, **Parse** button, and **Fetch** button.
- Status** field.
- Bottom navigation buttons: **Save**, **Cancel**, **Undo**, **Notes**, and **Help**.

4. Click the **Select User** list and select User #1, User #2, or Local User; and select the appropriate Security Level. If you select Local User, you can configure a node-specific username, authentication and privacy settings, and context as necessary.
5. Select the new protocol from the **Authentication** list box.
6. Click **Save**.

Note: You can change the protocol for one or more nodes from the Node List window. Right-click one or more nodes, click **Authentication**, and then select the protocol you want for the nodes. Polling is halted for all selected nodes during this change.

Classifying the SNMP Version Configured on Nodes

A node must have SNMP version information before NerveCenter can poll the node or process a trap from the node. NerveCenter enables you to obtain the SNMP version for a node and classify the node with that version. This is required when you don't know the SNMP version for a node or when NerveCenter receives its nodes from a Network Management Platform (NMP). When NerveCenter receives nodes from OpenView, NerveCenter deems the SNMP version for these nodes to be version 1.

A node must already exist in the database before it can be classified. To classify a node as SNMPv3, the agent must have an initial user configured for discovery. For details, see [Managing SNMP Settings](#) in *Managing NerveCenter*.

For a detailed study of classification, see [The Need for Node Classification](#) in *Managing NerveCenter*.

There are three ways in which NerveCenter classifies nodes:

- Enable auto-classification of nodes. If auto-classification is enabled, when NerveCenter adds nodes to its database (discovered from a trap, added from a NMP, or imported from another NerveCenter), any nodes without version information are classified at the highest possible level up to the maximum version specified in NerveCenter Administrator. NerveCenter does not attempt auto-classification for nodes that you add manually in Client.

For details, refer to [Managing Node Data](#) in *Managing NerveCenter*.

- Manually classify SNMP version for one or more nodes. NerveCenter attempts to classify one or more nodes at the highest level up to the maximum version specified in NerveCenter Administrator.

For details, see [Classifying the SNMP Version for One or More Nodes Manually](#) on page 109 in *Managing NerveCenter*.

- Manually classify all nodes in the Client's Node List. NerveCenter attempts to classify all nodes in its database at the highest level up to the maximum version specified in NerveCenter Administrator.

For details, see [Classifying the SNMP Version for All Nodes Manually](#) on page 110.

Note: You can also manually confirm the SNMP version defined for a node. When you use this option, NerveCenter attempts to poll a node using the version specified for the node. The maximum classified version configured in NerveCenter Administrator has no effect on this operation. For details, see [Confirming the SNMP Version for a Node](#) on page 111.

If NerveCenter classifies a node as SNMPv3, NerveCenter assigns a default security level for communicating with the node. The default security level is NoAuthNoPriv. For details about changing the security level, see [Changing the Security Level of an SNMPv3 Node](#) on page 102.

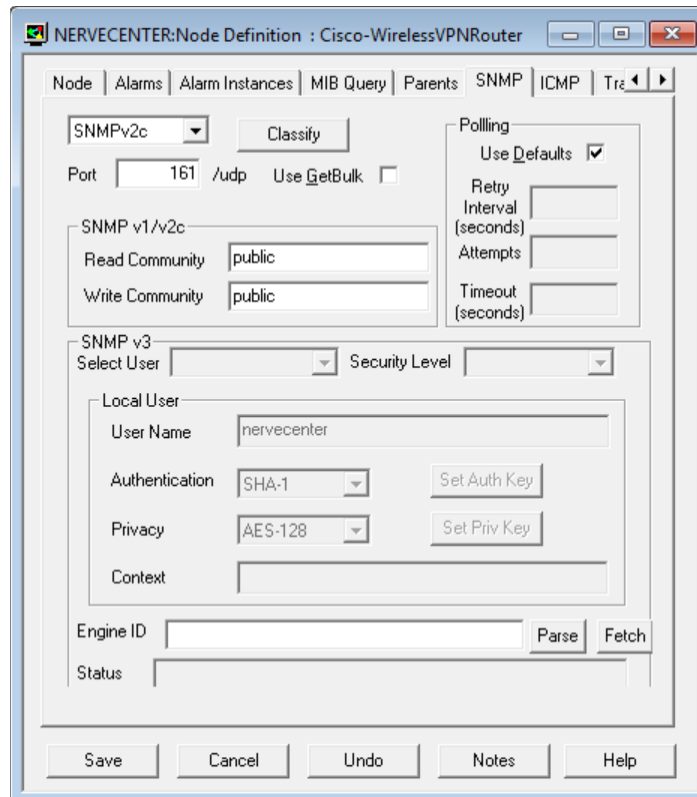
Caution: If NerveCenter fails to classify the node, then the version of the node is set to "Unknown." NerveCenter does not poll nodes or process traps from nodes whose SNMP version is Unknown.

For more information about classification, see also:

- [When NerveCenter Classifies Node SNMP Versions on page 114](#)
- [How NerveCenter Classifies a Node SNMP Versions on page 115](#)

SNMP version classification is also important for using the GetBulk function.

You can disable NerveCenter's per-node default of using GetBulk. To block NerveCenter from using GetBulk for a node, find it in the Node List, open it and select the 'SNMP' tab. There, uncheck the 'Use GetBulk' checkbox and then select Save.



NerveCenter utilizes the SNMP `getbulk` operation. `Getbulk` operations allow NerveCenter to retrieve large portions of a table in a single response, instead of making repeated `getnext` requests. This increases NerveCenter's polling performance on nodes possessing MIB tables with a large number of instances, such as a managed device with many ports or interfaces (for example, multiple `ifEntry` table rows).

You do not need to make any changes to take advantage of the `getbulk` functionality, though certain criteria must be met for NerveCenter server to use `getbulk` instead of `getnext`.

- `Getbulk` operations are used when a polled Node's SNMP version is classified as v2c or v3.
- The Property being polled must be associated with a MIB table (as opposed to a singleton object).
- The Property Group's Property must not have a Filter applied.
- Instance-scope and Subject-scope alarm instances that have transitioned out of ground utilize standard individual polls to allow flexible state transitions for each unique instance.

With these restrictions in mind, you can employ several techniques to optimize the performance of NerveCenter polling:

- Classify Node versions as at least v2c if they support that level of SNMP
- Utilize Property Filters only when eliminating a large range of instances from a large table
- Design Alarm models that do not transition out of ground for trivial events.

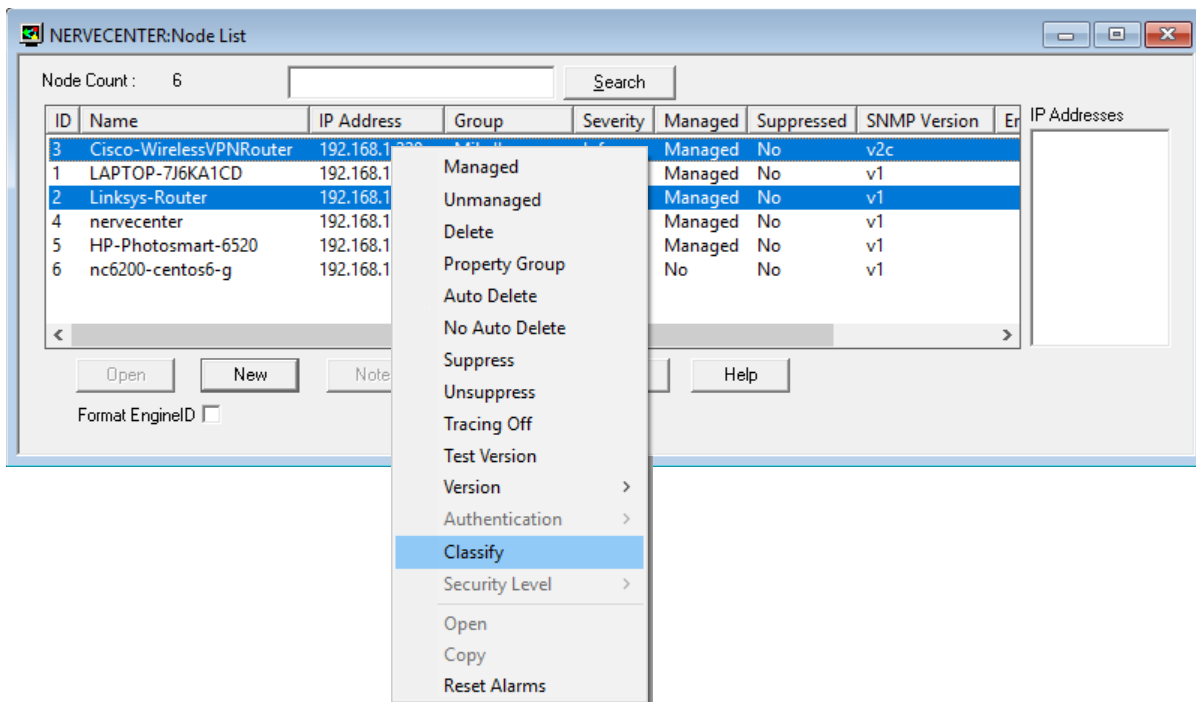
Classifying the SNMP Version for One or More Nodes Manually

Follow the procedure below to classify the SNMP version for one or more nodes manually. When using this method, NerveCenter attempts to classify the selected nodes at the highest level up to the maximum version specified in NerveCenter Administrator.

TO CHANGE THE AUTHENTICATION PROTOCOL USED TO MANAGE AN SNMPV3 NODE

1. From the client's **Admin** menu, select **Node List**.

The Node List window is displayed.



2. In the Node List window, select **New** if defining a new node, or select the node and then **Open** to change an existing node.
The Node Definition window appears.
 3. Right-click the node or nodes you want to classify and select **Classify**.
NerveCenter attempts to classify the SNMP version on the nodes up to the highest level specified in NerveCenter Administrator.
-

Classifying the SNMP Version for All Nodes Manually

NerveCenter Client can attempt to classify nodes' SNMP versions, at the highest level to the maximum specified in NerveCenter Administrator.

TO CLASSIFY NODES MANUALLY

- From the client's **Admin** menu, select **Classify All Nodes**.
NerveCenter attempts to classify the SNMP version on all nodes up to the level specified.
-

Confirming the SNMP Version for a Node

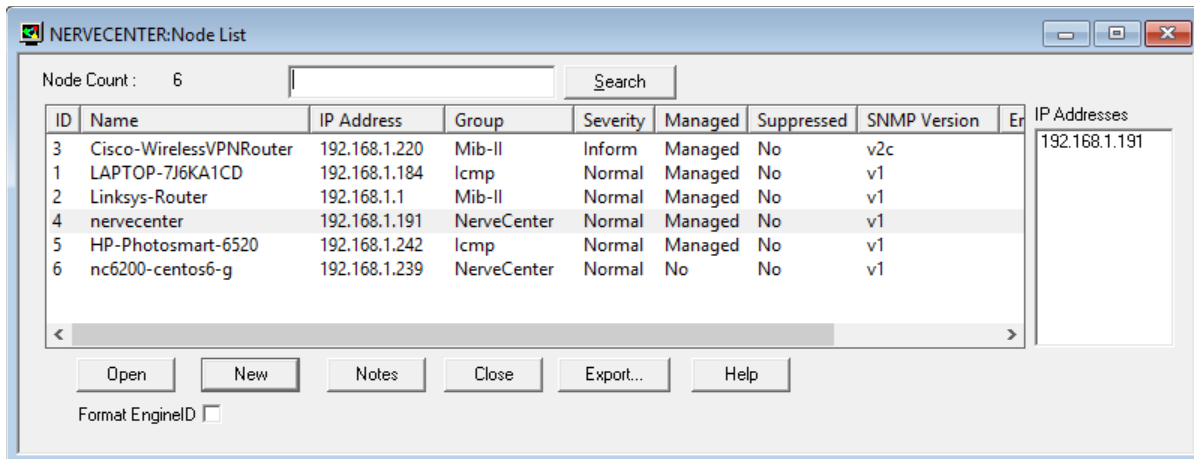
You can verify the SNMP version that NerveCenter has configured for any particular node, which is useful when manually defining a node to be added to the node list.

With this option, NerveCenter attempts to poll the node using the version specified; the maximum classified version configured in NerveCenter Administrator does not apply to this classification method. For example, if the maximum classification value is v2c and you have set the version for a particular node to SNMPv3, you can still confirm SNMPv3 communication using this method.

TO CONFIRM A NODE'S SNMP VERSION

1. From the client's **Admin** menu, select **Node List**.

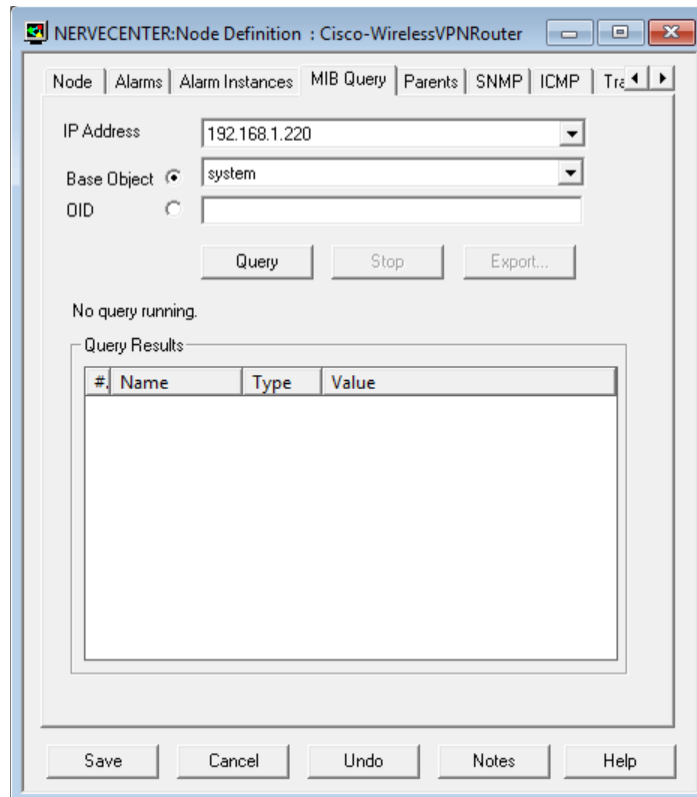
The Node List window is displayed.



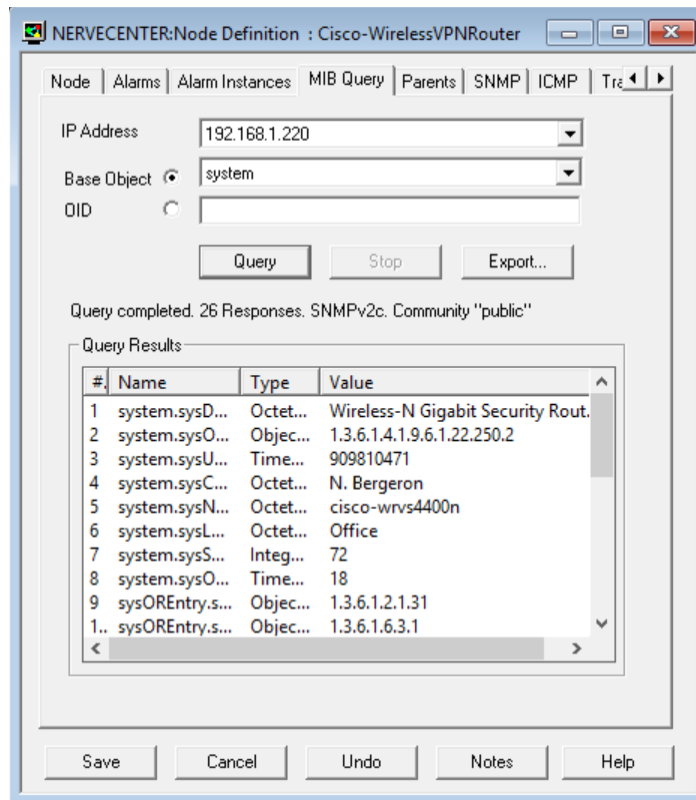
- In the Node List window, select **New** if defining a new node, or select the node and then **Open** to change an existing node.

The Node Definition window appears.

- Select the **MIB Query** tab.



4. Select the **Query** button.



NerveCenter attempts to communicate with the node using the SNMP version specified in the SNMP Version field on the SNMP tab. See [Testing SNMPv1 and v2c agents below](#) and [Testing SNMPv3 agents below](#) for details.

Testing SNMPv1 and v2c agents

To test the agent on a node configured in NerveCenter with SNMP version 1 or 2c, the Test Version poll sends the agent an SNMP GetRequest for the system description, sysDescr.0. This operation is similar to the GetRequest issued by clicking the Get button on the Query Node tab of a node's definition window.

Testing SNMPv3 agents

To test the agent on a node configured in NerveCenter with SNMPv3, the Test Version poll issues GetRequest messages for the following:

- Engine ID for a node
- Boots/timeticks if the security level on the node is either AuthNoPriv or AuthPriv
- sysObjectID for the node

To establish communication, NerveCenter sends a `GetRequest` for the node's `sysObjectID`. Before sending this `GetRequest`, however, NerveCenter first requires engine information such as `engineID`, `engineBoots`, and `timeTicks`. If this information is not known to NerveCenter, NerveCenter cannot send a request to the agent.

NerveCenter must obtain engine information in the following cases:

- When the SNMPv3 node has an `'v3InitFail'` error status
This status indicates that the `engineID` for that node is not available to NerveCenter. NerveCenter first obtains the engine ID. Then, if the security level for the node is other than `NoAuthNoPriv`, NerveCenter obtains the `boots` and `timeTicks`.
- When the SNMPv3 node has an error status of `'TimeSyncFail.'`
This status indicates that the `engineBoots` and `timeTicks` for that node are not available to NerveCenter.
- When someone has changed the Authentication and Privacy passwords in NerveCenter Administrator but did not update the passwords on the SNMPv3 agent.
You must change the passwords on the agent and run the `TestVersionPoll` to restore proper communication.

After obtaining the engine information, NerveCenter can send the `sysObjectID` request.

When NerveCenter Classifies Node SNMP Versions

There are two main ways that NerveCenter classifies nodes:

- **On demand**—You can issue a `classify` command in NerveCenter Client to classify one, several, or all nodes in the database.
- **Automatically**—You can set up auto-classification in NerveCenter Administrator. Then, when NerveCenter adds nodes to its database (discovered from a trap, added from a network management platform, or imported from another NerveCenter), any nodes without version information are classified at the highest possible level. NerveCenter does not attempt auto-classification for nodes that you add manually in Client. Refer to [SNMP Auto and Manual Classification Settings](#) in *Managing NerveCenter* for details about auto-classification.

When you enable auto-classification, NerveCenter attempts auto-classification in the following instances:

- A node is added through a node file either from `importutil` or from the Client, and the node does not have a version or has the version "Unknown." This would happen, for example, if you were importing the node from a previous version of NerveCenter.
- A node is imported from another NerveCenter Server, and the node does not have a version or has the version "Unknown."
- A node is added from a trap, and the node's version is not v3. NerveCenter needs to verify whether these nodes are v1 or v2. If the trap indicates v3, NerveCenter does not need any further verification.

- NerveCenter is co-resident with network management platform and the platform sends nodes to NerveCenter. All nodes added from a NMP are v1 by default.

Note: NerveCenter does not attempt to auto-classify nodes added manually in Client.

Disabling auto-classification in Administrator prevents auto-classification for all these cases. If you choose to disable auto-classification, bear in mind that NerveCenter does not poll nodes whose SNMP version is unknown. (You can still classify nodes manually in NerveCenter Client using the available commands.)

How NerveCenter Classifies a Node SNMP Versions

There are two main ways that NerveCenter classifies nodes:

- **Manually**—You can issue a classify command in NerveCenter Client to classify one, several, or all nodes in the database.
- **Automatically**—NerveCenter can be configured to classify nodes when they are added to its database (discovered from a trap, added from a network management platform, or imported from another NerveCenter). Refer to [SNMP Auto and Manual Classification Settings](#) in Managing NerveCenter for details about auto-classification.

Following is a summary of classification.

Each time NerveCenter attempts to classify a node, NerveCenter sends a series of classification requests (GetRequest messages) to the node. NerveCenter classifies the node based on the responses to these requests. Each request corresponds to an SNMP version—either v1, v2c, or v3.

While classifying a node, NerveCenter attempts to detect the maximum supported version on the agent up to a maximum specified version, which you can configure in NerveCenter Administrator. So, for example, if you set a maximum classification version of v2c, NerveCenter never attempts to classify nodes any higher than v2c. (However, you can manually specify any version for a node and then test communication with the agent using that version. See [Manually Changing the SNMP Version Used to Manage a Node on page 99](#) for details.)

Based on the response to its messages, NerveCenter changes its SNMP version setting for the node.

Caution: Note the following about node classification: 1) If NerveCenter fails to classify the node, then the version of the node is set to “Unknown.” NerveCenter cannot poll a node with an unknown version; 2) A node must have correct version information, either supplied manually by the user or obtained via classification, before NerveCenter can successfully poll the node or process a trap from the node.

SNMP version 3 is an extension of SNMP that addresses security and administration.

Overview of NerveCenter SNMPv3 Support

NerveCenter support for SNMPv3 includes new data types and enhanced security for communication. SNMP v1 and v2c rely on community names for authentication. SNMPv3 enhances authentication and expands its services to include privacy. SNMPv3 also expands on the earlier concept of MIB views to control access to management information by using a View-based Access Control Model (VACM) to determine a user's access level for viewing MIB data.

Following are highlights of NerveCenter support for SNMPv3:

- Before NerveCenter can discover SNMPv3 agents on nodes, the nodes must have an initial user configured for discovery.

See [Configuring an Initial User for Discovering an SNMPv3 Agent](#) in Managing NerveCenter.

Refer to [Confirming the SNMP Version for a Node on page 111](#) for details about testing communication with a node using the NerveCenter Test Version poll.
- NerveCenter communicates (sends polls) with SNMPv3 agents on behalf of a specified NerveCenter user (the global User #1 or User #2 accounts, or a node-specific user). Before NerveCenter can poll SNMPv3 agents, the agents must be configured to support the user, security level, and potentially the context.

See [Configuring an SNMPv3 Agent for NerveCenter](#) in Managing NerveCenter.

See [Configuring SNMPv3 Security Settings](#) in Managing NerveCenter.
- NerveCenter supports three security levels for communicating with SNMPv3 agents. By default, NerveCenter sets the security level to noAuthNoPriv, which means the v3 agent sends and receives messages without authentication or encryption.

See [NerveCenter Support for SNMPv3 Security on the next page](#) for details about security.

Refer to [Changing the Security Level of an SNMPv3 Node on page 102](#) for details about setting a node's security level.
- The authentication and privacy protocols require specialized authentication and privacy keys, which are generated from the corresponding passwords. You can change these passwords in NerveCenter, thereby changing the keys.

See [NerveCenter Support for SNMPv3 Digest Keys and Passwords on page 119](#).

See [Configuring SNMPv3 Security Settings](#) in Managing NerveCenter.

- NerveCenter supports the HMAC-MD5-96 (MD5) or HMAC-SHA-96 (SHA) protocols for authentication on a per-node basis, and DES, 3-DES, AES-128, AES-192, or AES-256 as privacy protocols. If you change an agent's authentication protocol, you must likewise configure NerveCenter to use that protocol to manage the corresponding node in its database.

Refer to [Changing the Authentication Protocol for an SNMPv3 Node on page 105](#) for details about changing the authentication protocol used by NerveCenter for an agent.

- A node must have SNMP version information before NerveCenter can poll or process a trap from that node. NerveCenter can discover the node version both automatically or manually. If auto-classification is enabled, then a newly added node (e.g., discovered from a trap, added from a platform such as HP OpenView, imported from another NerveCenter installation) will be classified at the highest level possible.

Note: NerveCenter auto-classification is disabled by default. You must enable it before NerveCenter can classify nodes added to its database.

See [SNMP Auto and Manual Classification Settings](#) in Managing NerveCenter.

Refer to [Classifying the SNMP Version Configured on Nodes on page 107](#) for details about classifying nodes manually.

- SNMPv3 operations are logged to a file so that you can follow the progress of v3 activities. The log includes information about activities (e.g., a key change initiated by the user) as well as errors that occur while NerveCenter attempts to perform the activities.

See [Viewing the SNMPv3 Operations Log on page 120](#).

See [SNMP Error Status on page 124](#) for information about SNMPv3 errors.

NerveCenter Support for SNMPv3 Security

SNMPv3 enables devices to communicate in a secure fashion using message authentication to validate users and encryption to provide communication privacy. SNMPv3 provides a User-based Security Model (USM) to establish authentication and secrecy.

SNMPv3 nodes can have one of the following security levels:

- **NoAuthNoPriv** — Neither message authentication nor encryption is used while communicating with the agent. No passwords are required.
- **AuthNoPriv** — Message authentication is used without encryption while communicating with the agent. An authentication protocol and password are required to be set up in agreement on the device and within NerveCenter's definition for the respective node. In NerveCenter the authentication protocol and password to be used can be set to either of the global protocols and passwords User #1 or User #2, or can be set on a per-node basis.
- **AuthPriv** — Both authentication and encryption are used when communicating with the agent. Both the authentication and privacy specifics are required to be set up in agreement on the device and within NerveCenter's definition for the respective node. In NerveCenter the protocols and passwords can be set to either of the global User #1 or User #2 definitions, or can be set on a per-node basis.

Communication between any two SNMPv3 entities takes place on behalf of a uniquely identified domain user. The security level used for this communication defines the security services — message authentication and encryption — used while exchanging data. NerveCenter communicates with SNMPv3 nodes on behalf of the NerveCenter poll user in the poll context.

If you do not specify a security level for an SNMPv3 node, NerveCenter uses the NoAuthNoPriv security level by default, which means that message authentication and encryption services are not used for data exchange with the node.

Note: The NerveCenter poll users (User #1 and User #2), contexts, and the authentication and privacy passwords can be changed in NerveCenter Administrator.

The node-specific information (such as version, security level, and authentication protocol) used to poll each SNMPv3 node is configured in NerveCenter Client. A node-specific poll user and associated credentials can be managed from the SNMP tab on the Node screen (see ["Changing the Security Level of an SNMPv3 Node"](#) on page 102).

NerveCenter Support for SNMPv3 Digest Keys and Passwords

SNMPv3 allows two devices to communicate in a secure fashion using message authentication and encryption to ensure secrecy. In any SNMPv3 communication, one of the two communicating entities plays a role of authoritative entity for the communication, and communication is performed on behalf of a unique user within the management domain.

The sender of a secure message attaches a code, or digest, for authentication and encrypts the message to ensure privacy. To generate this digest, the sender uses an authentication key at the authoritative entity of the user on whose behalf communication takes place. Similarly, to encrypt a message, the sender uses a privacy key at the authoritative entity of the user on whose behalf communication takes place. These keys are generated from the authentication password and privacy password, respectively.

SNMPv3 specifications have defined a localized key-generation scheme. For every user, the authentication key at every SNMPv3 entity is a function of the snmpEngineID of that entity, the user's authentication password, and the authentication protocol. For every user, the privacy key at every SNMPv3 entity is a function of the snmpEngineID of that entity, the user's privacy password, and the privacy protocol. NerveCenter supports this localized key-generation scheme.

NerveCenter communicates with SNMPv3 nodes on behalf of a NerveCenter user (User #1, User #2, or a local, node-specific user). NerveCenter needs to know the authentication and privacy passwords for this user to generate the keys required for secure communication. Whenever NerveCenter learns the snmpEngineID of a newly discovered SNMPv3 agent with a security level other than NoAuthNoPriv, NerveCenter generates these keys for the NerveCenter poll user on that agent:

- If authentication is required (a security level of AuthNoPriv is specified for the node), the sender uses the authentication key to generate the digest for the message, which is appended to the message.
- If encryption is required (a security level of AuthPriv is specified for the node), the sender uses the privacy key to generate the digest for the message. For this security level, only the privacy digest is required; privacy assumes authentication, and you cannot have encryption without authentication.

On receipt of a secure message, a receiver does the following:

- Separates the message from the digest (authentication or privacy).
- Uses the corresponding key from its local store to generate the message's local digest copy.
- Compares the local digest with the one received in the message. If the two digests match, the recipient authenticates or decrypts the message using the corresponding local key. If they do not match (indicating a lack of authentication), the recipient discards the message.
- The recipient reads and processes the message.

Viewing the SNMPv3 Operations Log

Whenever a NerveCenter Server receives a request for an SNMPv3 operation or an error occurs while attempting to perform an SNMPv3 operation (e.g., v3 initialization fails), the NerveCenter Server logs a message to V3Messages.log, which resides in the NerveCenter installation log directory on the NerveCenter Server host machine. The file contains messages about SNMPv3 operations and errors resulting from requests that originate with any connected NerveCenter Clients, Administrators, and Command Line interfaces.

After logging the error, the NerveCenter Server notifies all connected NerveCenter Clients and Administrators in the following ways:

- If you are logged on to the NerveCenter Client or Administrator that initiated the operation that caused an error condition, NerveCenter displays the error that was logged.
- If you are logged on to a NerveCenter Client or Administrator that did not initiate the error condition, a red icon appears in the status bar; double-click the icon to display the NerveCenter Server with the SNMPv3 error. If your Client or Administrator is connected to more than one Server, the dialog box lists all servers that currently have an error condition.

When your NerveCenter Client or Administrator displays a dialog box with an error condition, you can do either of the following:

- Acknowledge the error condition by "signing the log." When you sign the log, NerveCenter notes that in the log file and changes the red icon back to green for all connected Clients and Administrators.
- Dismiss the dialog box without acknowledging the error condition, in which case only the icon in your Client or Administrator turns green. The icon remains red for all other connected Clients and Administrators to signal that the NerveCenter Server has an unacknowledged/unsigned error. Moreover, the Server does not indicate acknowledgment in the log file.

You must have administrator rights to initiate an SNMPv3 operation that can result in an error or to acknowledge a logged error condition. If you are logged on with only user rights, you can dismiss the error dialog box but not acknowledge an error condition.

Whether you acknowledge or dismiss the error, all messages remain in the V3Messages.log.

For more information, refer to the following topics:

- ["Signing a Log for SNMPv3 Errors Associated with Your Client" on the facing page](#)
- ["Signing a Log for SNMPv3 Errors Associated with a Remote Client or Administrator" on page 122](#)
- ["Viewing the SNMPv3 Operations Log" on page 123](#)

Signing a Log for SNMPv3 Errors Associated with Your Client

Whenever an SNMPv3 operation is requested or an error occurs while attempting an SNMPv3 operation, the NerveCenter Server logs a message to V3Messages.log. If you are logged in to the NerveCenter Client that initiated the logged request, NerveCenter displays a dialog box with that error.

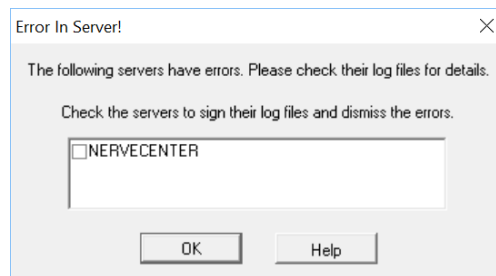


Figure 23: Operations Log Error in Server Dialog Box for Your Client

Users with administrator rights can acknowledge a logged condition from NerveCenter Client by signing the Operations log. Signing the log causes the icon to turn green in all connected Clients/Administrators. You can also dismiss the dialog box without acknowledging the error condition. If you are logged on with user rights rather than administrator rights, your only option is to dismiss the dialog box; you cannot sign the Operations log..

TO SIGN THE OPERATIONS LOG

1. After viewing the message that NerveCenter displays on your screen, check the Sign the log and dismiss errors checkbox.
2. Click **OK**.
The icon in the Status Bar turns green for all Clients or Administrators connected to the designated NerveCenter Server. You can later view this message again in the Operations log.

This V3Messages.log file resides in the NerveCenter installation log directory. The file can be viewed in a text editor or word processor.

TO DISMISS THE ERROR IN SERVER DIALOG BOX

- Click **OK** without checking the checkbox.
In this case, only the icon in your Client turns green. For all other connected Clients and Administrators, the icon remains red and signals to those modules that the NerveCenter Server has some error that remains unacknowledged.

Signing a Log for SNMPv3 Errors Associated with a Remote Client or Administrator

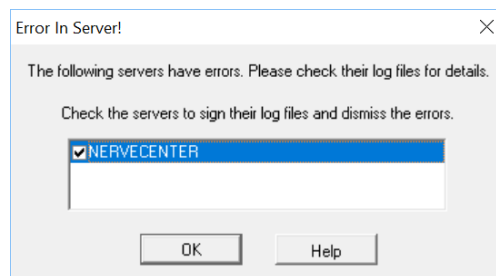
Whenever an error occurs while attempting an SNMPv3 operation, the NerveCenter Server logs a message to V3Messages.log. If you are logged on to a remote NerveCenter Client (one that did not initiate the error condition), the status bar displays a red icon.

Users with administrator rights can acknowledge a logged condition from NerveCenter Client by signing the Operations log. Signing the log causes the status icon to turn green in all connected Clients/Administrators.

You can also dismiss the dialog box without acknowledging the error condition. If you are logged on with user rights rather than administrator rights, your only option is to dismiss the dialog box; you cannot sign the Operations log..

TO SIGN THE OPERATIONS LOG

1. Double-click the red icon in the Status Bar.
The Error In Server dialog box is displayed.



2. Check the NerveCenter Server or Servers for which you want to sign the log.
3. Click **OK**.

The icon in the Status Bar turns green for all Clients or Administrators connected to the servers you checked. At a suitable time, you can open the Operations log and view the new message. This file, named V3Messages.log, resides in the NerveCenter installation log directory. The file can be viewed in a text editor or word processor.

TO DISMISS THE ERROR IN SERVER DIALOG BOX

1. Double-click the red icon in the Status Bar.
The Error In Server dialog box is displayed.
2. Click **OK** without checking any of the checkboxes.
In this case, only the icon in your Client turns green. For all other connected Clients and Administrators, the icon remains red and signals to those modules that the NerveCenter Server has some error that remains unacknowledged.

Viewing the SNMPv3 Operations Log

Whenever an SNMPv3 operation is requested or an error occurs while attempting the operation, the NerveCenter Server logs a message to the V3Messages.log file, which resides in the NerveCenter installation log directory on the NerveCenter Server host machine.

The file can be viewed in a text editor or word processor. As NerveCenter adds more messages to the file, the file continues to grow until you manually remove old messages.

The log entries resemble the following:

```
06/20/2017 09:26:29 Tue - Event ID : NC_SERVER; Category ID : NC_
THREAD_V3OP; /
      Error Status : AutoClassifyFail; Error while communicating
using SNMPv1 for 10.52.174.51 /
      because of : NC_PORT_UNREACHABLE;
```

Following are the fields in the log:

Table 12: Fields in the Operations Log

Field	Description
Date/Time	Date and time the record was logged. The format is month/day/year, hour/minute/second, and day (for example, 12/16/2017 11:32:29 Sat).
EventID	This always NC_SERVER.
CategoryID	Name of the thread where the event occurred.
Error Status	One of several error status strings. See <i>Error Status</i> for a description of SNMPv3 error status messages and which ones cause polling to stop for a node.
Error Description	Details of the error or operation.

SNMP Error Status

When NerveCenter is unable to complete an SNMP operation on a node, the error status is displayed in the Node List (NerveCenter Client and Web Client) and in the SNMP tab of the node's definition window (NerveCenter Client). Figure 24 shows the Node List window in the Client.

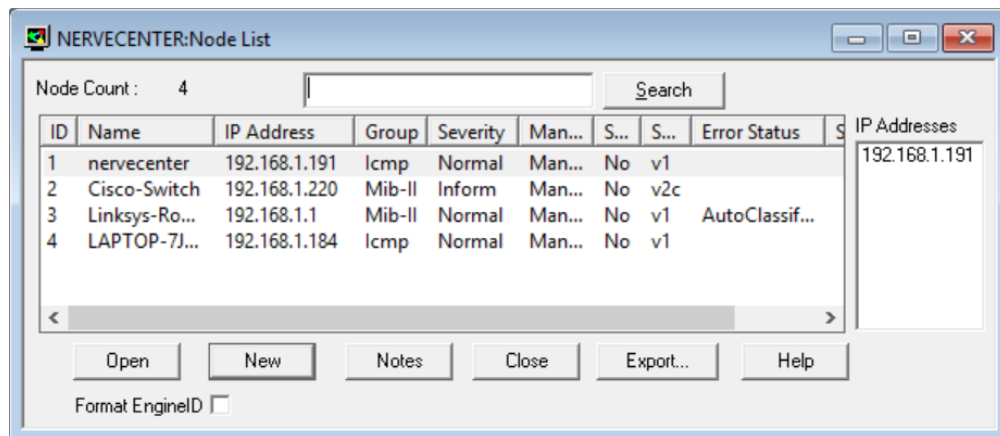


Figure 24: Node List Window

Though most of the error strings correspond to SNMPv3 errors, some are applicable for v1 and v2c errors as well. These are noted in the descriptions below.

Sometimes error conditions can be corrected simply by running the SNMP Test Version poll. Others may require configuration changes to the node's SNMP agent. After changing the configuration of an SNMP agent, always test communication with the node in NerveCenter Client prior to polling the node.

The following list describes each possible SNMP error status.

- **V3InitFail** – An attempt to get the snmpEngine ID of an SNMPv3 agent failed or the SNMPv3 configuration defined for that node is causing a failure at the SNMPv3 communication layer. This can occur either when NerveCenter first attempts to poll the node using the SNMPv3 configuration or at any point when the SNMP agent changes its SNMPv3 configuration. For all of these cases, the V3InitFail is augmented by one of the following values in the SNMPv3 Status field (NerveCenter Client):
 - ConfigurationError – The node's SNMP definition is incomplete with respect to its Security Level. This status is discovered and reported by NerveCenter before issuing an SNMPv3 request to an SNMP Agent.
 - Operator intervention is required. The node's SNMP v3 definition must contain a User Name regardless of the Security Level — AuthNoPriv requires an Authentication Protocol and Password; AuthPriv requires Authentication and Privacy Protocols and passwords for each.
 - UnknownUsername – The SNMP Agent reports that the SNMPv3 User Name being sent by NerveCenter is not one of the user names that it has been configured to handle.
 - UnknownContext – The SNMP Agent reports that the SNMPv3 Context being sent by NerveCenter is not appropriate. Many SNMP Agents do not report this value, even if it is the underlying issue. Instead, the SNMP Agent may not issue any response and the operation will time out.

- **UnavailableContext** – The SNMP Agent reports that the SNMPv3 Context being sent by NerveCenter is known but inapplicable to the operation (poll, discovery, or classification) being attempted. Many SNMP Agents do not report this value, even if it is the underlying issue. Instead, the SNMP Agent may not issue any response and the operation will time out.
- **UnsupportedSecLevel** – The SNMP Agent reports that it cannot handle the Security Level defined in a request issued to it by NerveCenter.
- **UnknownEngineID** – Either NerveCenter’s SNMP Stack or the SNMP Agent is reporting an issue with the `snmpEngineID` used for SNMP v3 communication. This can occur if the `snmpEngineID` is changed on the SNMP Agent during polling.
- **IncorrectAuthPasskey** – The SNMP Agent reports that the Authentication passkey (digest) being issued by NerveCenter is not correct. This generally occurs in one of two cases: 1) An incorrect password was entered either on the SNMP Agent or in NerveCenter, or 2) The password was entered correctly at both ends, but the selected Authentication protocol is mismatched between the SNMP Agent and NerveCenter.
- **ClassifyFail** – An attempt to obtain the node’s SNMP version failed during a classification attempt. The node’s version will be set to “Unknown” and it will not be polled. You can manually change the version or try to classify the node again.
- **AutoClassifyFail** – An auto-classification attempt failed to obtain the node’s version. The node’s version will be changed to “Unknown” and it will not be polled. You can manually change the version or try to classify the node again.

Note: `ClassifyFail` and `AutoClassifyFail` status values are not limited to SNMPv3 agents. If NerveCenter attempts to classify an agent and fails for some reason (e.g., the agent is down), NerveCenter will mark the node with `ClassifyFail` or `AutoClassifyFail` regardless of the SNMP version supported on the agent.

- **TestVersionFail** – An attempt to poll the SNMP agent failed. The Test Version poll sends a `GetRequest` message for a node based on the SNMP version configured for that node.

If the Test Version poll fails, polling will not happen for this node. In that case, you may need to reconfigure the agent on this node. Then, try running the Test Version poll again (from a node’s definition window or the right-click menu in the node list).

Note: `TestVersionFail` is not limited to SNMPv3 agents. You can test the version of any SNMP agent with this feature.

- **Configuration Mismatch** – Indicates an SNMP trap was received but there is some problem with the configuration on the agent. If NerveCenter is unable to decode a trap due to some unspecified reason (e.g., unsupported authentication or privacy parameters on the agent or an incorrect NerveCenter user name), NerveCenter can receive the trap and add the node to its database if configured to discover nodes via traps. After adding the node to its database, however, NerveCenter assigns an error status of Configuration Mismatch.

Note: Any error that occurs during trap decoding always results in a Configuration Mismatch error.

- **TimeSyncFail** – An attempt to get the node’s snmpEngine boots/timeticks failed. Polling will continue for this node. If any polls successfully reach the node, the node responds with an “Out of time window” report PDU that contains the correct boots/timeticks, and NerveCenter can then update this information for the node. For the initial polls that generate the report PDU, the SNMP_NOT_IN_TIME_WINDOW trigger will be fired.
 - You can ignore this message, which simply indicates that NerveCenter is getting in sync with that node. You can recover from this error status by right-clicking the node in the Node List and selecting v3TestPoll. If the agent corresponding to the node is up, the test poll should be successful and clear the error message. The SNMPv3 Status field will be set to the following:
- **NotInTimeWindow** – This is the reply sent by the SNMP Agent or declared by NerveCenter’s SNMP stack upon investigating a request or response PDU wherein the SNMPv3 timestamp handling shows a time sync failure.

Using the SNMP Test Version Poll

When configuring an SNMP agent or if you encounter problems polling a node, you can test whether NerveCenter can communicate with the agent. NerveCenter provides an SNMP test poll that verifies communication with the node using the SNMP version specified for the node. If the agent is configured for SNMPv3, this poll helps you determine whether the agent is correctly configured for communication with NerveCenter.

If the poll fails to establish a connection for the specified SNMP version, a TestVersionFail error is displayed for the node, and polling will not happen for this node.

Testing SNMPv1 and v2c Agents

To test the agent on a node configured in NerveCenter with SNMP version 1 or 2c, the Test Version poll sends the agent an SNMP GetRequest for the system description. This operation is similar to the GetRequest issued by clicking the Get button on the Query Node tab of a node’s definition window.

Testing SNMPv3 Agents

To test the agent on a node configured in NerveCenter with SNMPv3, the Test Version poll issues GetRequest messages for the following:

- Engine ID for a node
- Boots/timeticks if the security level on the node is either AuthNoPriv or AuthPriv
- SysObjectID for the node

To establish communication, NerveCenter sends a GetRequest for the node’s sysobjectID. Before sending this GetRequest, however, NerveCenter first requires engine information such as engineID, engine boots, and time ticks. If this information is not known to NerveCenter, NerveCenter must send a request to the agent.

NerveCenter must obtain engine information in the following cases:

- When the SNMPv3 node has an 'v3InitFail' error status. This status indicates that the engineID for that node is not available to NerveCenter.
- NerveCenter first obtains the engine ID. Then, if the security level for the node is other than NoAuthNoPriv, NerveCenter obtains the boots and time ticks.
- When the SNMPv3 node has an error status of 'TimeSyncFail.' This status indicates that the engine boots and time ticks for that node are not available to NerveCenter.
- When someone has changed the Authentication and Privacy passwords in NerveCenter Administrator but did not update the passwords on the SNMPv3 agent. You must change the passwords on the agent and run the V3TestPoll to restore proper communication.

After obtaining the engine information, NerveCenter can send the SysObjectID request.

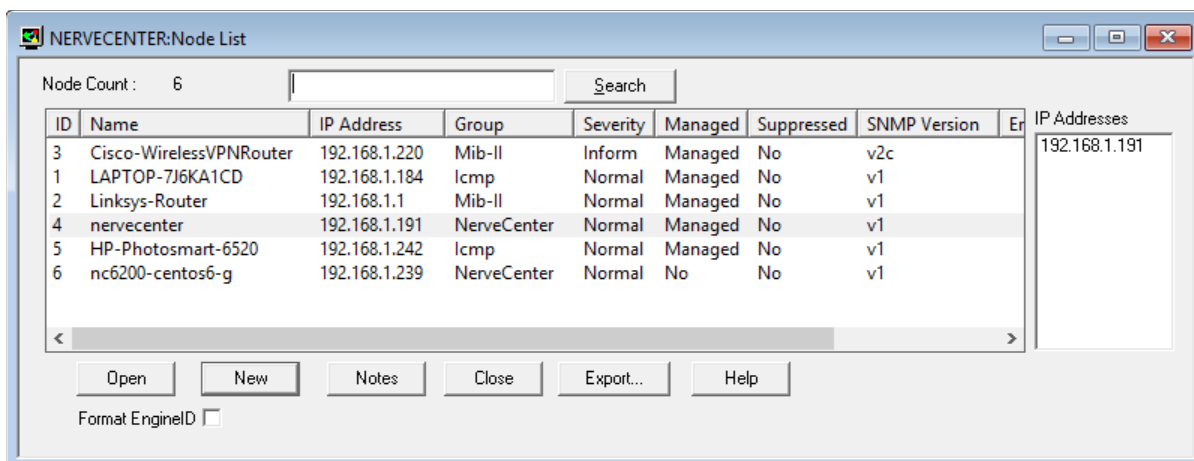
How To Use the Test Version Poll

Follow the steps below to verify communication with a node using the Test Version poll.

TO USE THE SNMP TEST VERSION POLL

1. From the client's **Admin** menu, select **Node List**.

The Node List window is displayed.



2. Right-click one or more nodes you want to test, then select **Test Version**.
-

Note: You can also issue this poll for a particular node by selecting the node in the list, clicking the Open button, and selecting **Test Version** in the SNMP tab.

The Status Bar indicates the status of the test. If the test fails to establish a connection for the specified SNMP version, a TestVersionFail error is displayed for the node.

Defining Property Groups and Properties

Recall that a property is a string, a property group is a container for properties, and property groups are assigned to nodes. In general, before NerveCenter will use a behavior model to manage a node, the following requirements must be met:

- The property of any poll in the behavior model must be in the node's property group.
- The name of the base object used in the poll condition of any poll in the behavior model must be in the node's property group.
- The property of any alarm in the behavior model must be in the node's property group.

This chapter concentrates on the mechanics of listing all existing property groups and properties, creating properties, creating property groups, and assigning property groups to nodes. The chapter concludes with a section that offers suggestions on how to use property groups effectively.

Listing Property Groups and Properties

When NerveCenter is first installed and the NerveCenter database is created, many property groups are loaded into the database. Before you begin creating new property groups, you should review these existing property groups and see if one of them meets your needs. Or perhaps you can create the property group you need by modifying an existing property group.

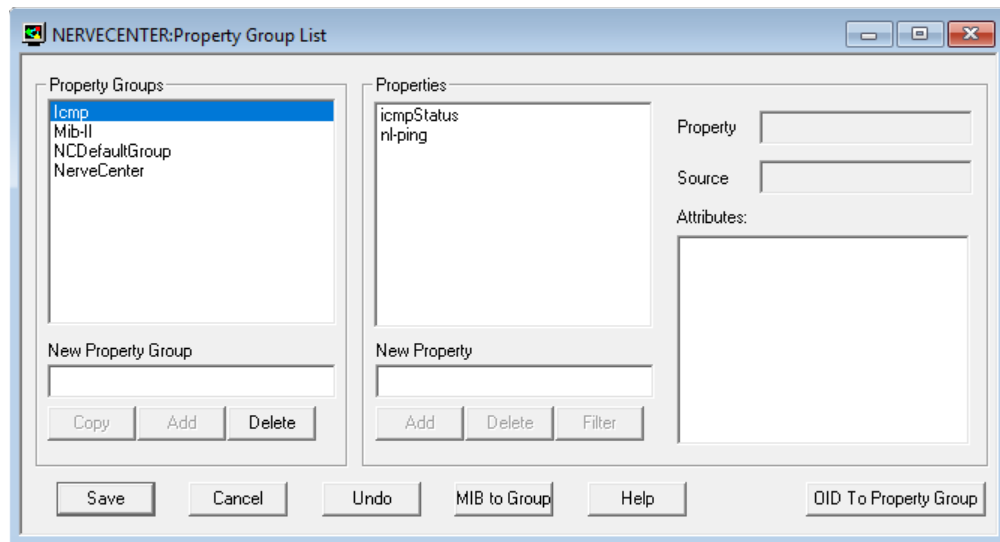
The following sections explain how to display a list of property groups and how to display a list of the properties in a property group:

- [Listing Property Groups on the next page](#)
- [Listing Properties on page 131](#)

Listing Property Groups

TO DISPLAY A LIST OF THE PROPERTY GROUPS CURRENTLY DEFINED IN THE DATABASE FOR THE ACTIVE SERVER

- From the client's **Admin** menu, choose **Property Group List**.
This action causes NerveCenter to display the Property Group List window.



The existing property groups are listed in alphabetical order in the Property Group list on the left side of the window.

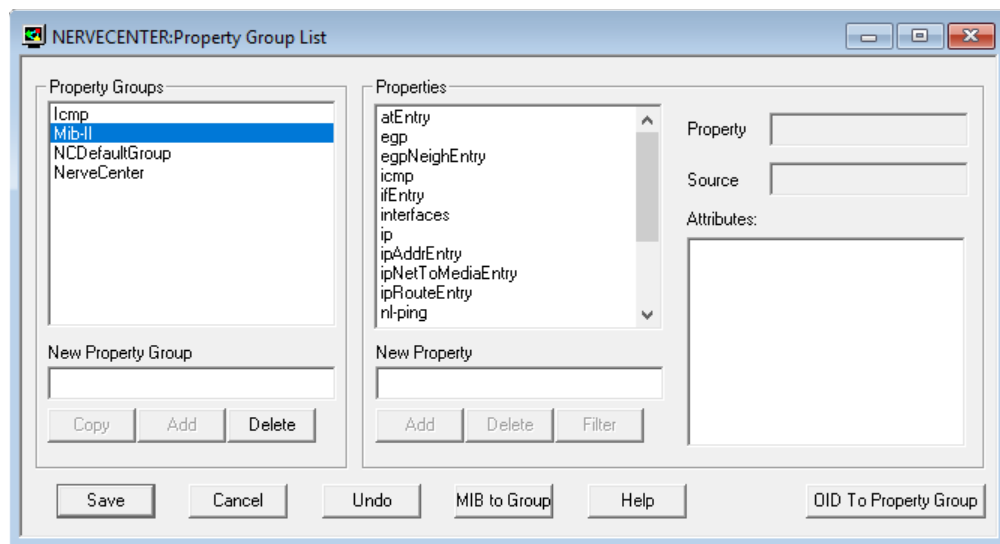
Listing Properties

You generally only display properties in the context of a property group. That is, you don't view all the properties defined in the database in a single list; you view a list of properties that belong to the same property group.

TO LIST THE PROPERTIES IN A PROPERTY GROUP

1. From the client's **Admin** menu, choose **Property Group List**.

NerveCenter displays the Property Group List window.



2. Select a property group from the **Property Group** list.

All of the properties belonging to that property group are listed in alphabetical order in the Property list on the right side of the window.

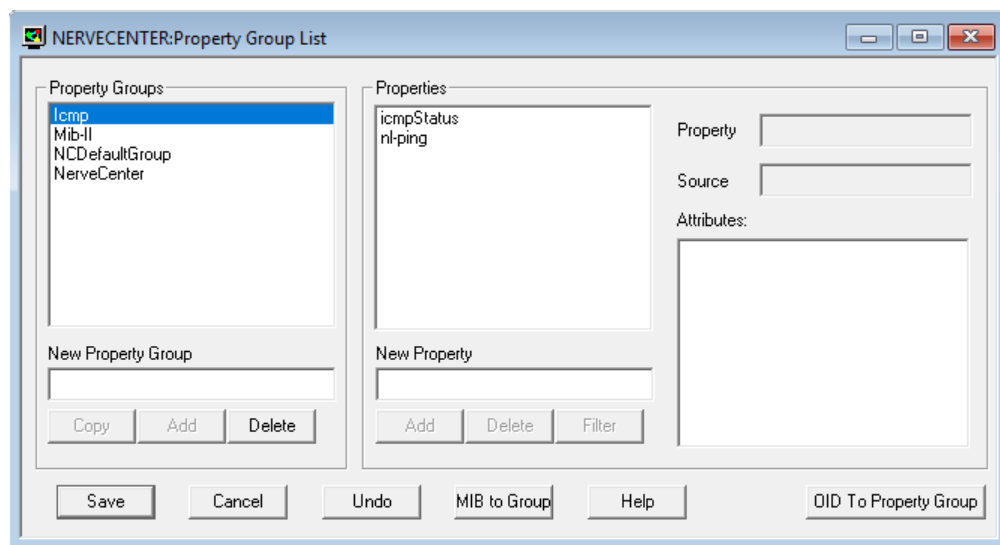
Creating a Property

If you design a new behavior model and intend it to manage a group of nodes that don't currently share a unique property, you must create a property to serve as that unique property. Because you must create this property in the context of an existing property group, you will probably need to create a property group before you create your property, as described in [Creating a New Property Group on the facing page](#). Once you've created both the property group and the property, you can assign the new property group to the nodes you want to manage with the new behavior model.

TO CREATE A PROPERTY

1. From the client's **Admin** menu, select **Property Group List**.

The Property Group List window is displayed.



2. Select a property group from the **Property Group** list.
Often you select a property group that you've created expressly to contain your new property. When you create the property, it will belong to this property group.
3. Type the name of the new property in the **New Property** text field.

Note: The maximum length for property names is 255 characters.

4. Select the **Add** button below the Property list.
The property is added to the Property list.
5. Select the **Save** button at the bottom of the window.

Creating a New Property Group

As you develop your network management strategy, you may need to create new property groups. For example, NerveCenter ships with a property group called Router that you can use to uniquely identify the routers on your network. However, suppose you decide that while some of your behavior models should apply to all routers, others should apply to either campus routers or backbone routers, but not both. To handle this problem, you might create two new property groups, CampusRouter and BackboneRouter. Each can be a copy of Router to which you add one unique property. For instance, you might add the property campusRouter to the property group CampusRouter and the property backboneRouter to the property group BackboneRouter. You could then assign these new property groups to the appropriate nodes.

There are three methods of creating a property group:

- You can base the new property group on an existing one. In this case, you copy an existing property group and then add one or more new properties to it. This is the technique used in the hypothetical example above.
- You can create a property group that contains the names of the base objects in one or more MIB definitions. This technique is useful when you add new hardware to your network and there is a special MIB defined for that hardware. Basing the property group on this MIB ensures that you'll meet one of the prerequisites for making the new device pollable: the base object used in the poll condition will be in the property group.
- You can create an empty property group and add properties to it one by one. Obviously, this option gives you the greatest flexibility, but it also is the most time consuming.

For further information on the three methods of creating a property group, see the sections listed below:

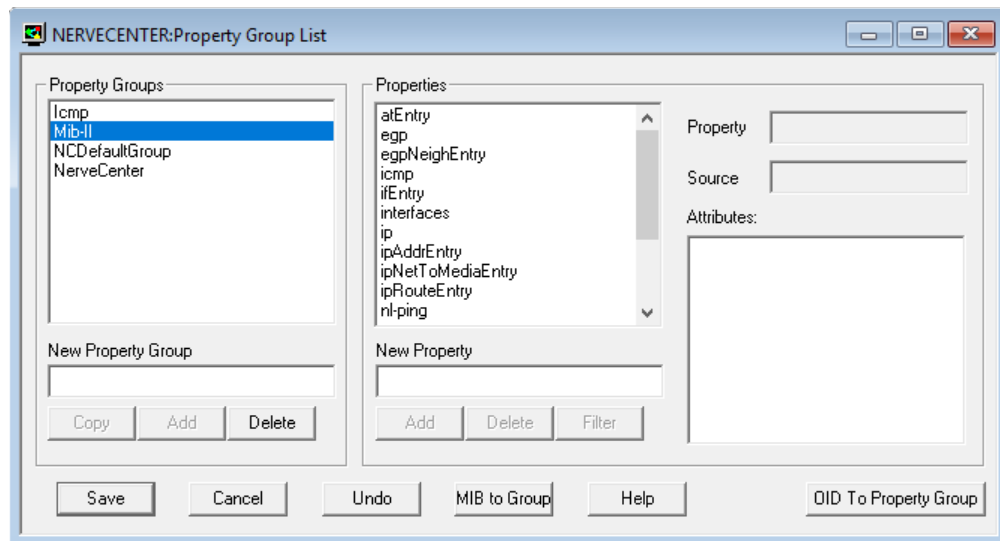
- [Based on an Existing Property Group on the next page](#)
- [Based on the Contents of MIBs on page 135](#)
- [Adding Properties Manually on page 138](#)

Based on an Existing Property Group

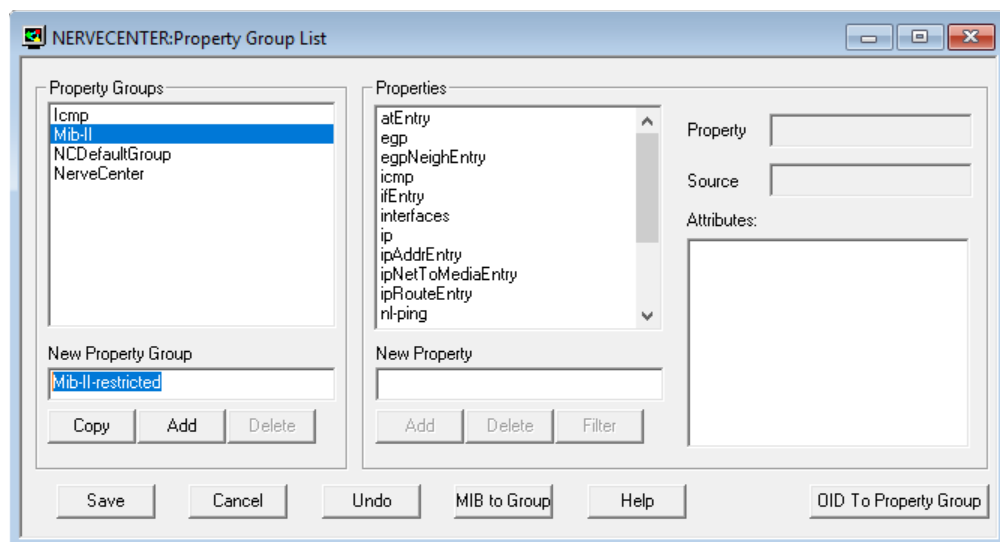
Earlier, we mentioned that you could create a property group for campus routers by copying the predefined property group Router, naming the copy CampusRouter, and adding to the new property group the unique property campusRouter.

TO CREATE A NEW PROPERTY GROUP BASED ON AN EXISTING ONE

1. From the client's **Admin** menu, select **Property Group List**.
The Property Group List window is displayed.



2. From the **Property Group** list, select the property group that you want to copy.
The properties contained in this property group are displayed in the Property list.
3. Type a name for the new property group in the **New Property Group** text field.



Note: The maximum length for property group names is 255 characters.

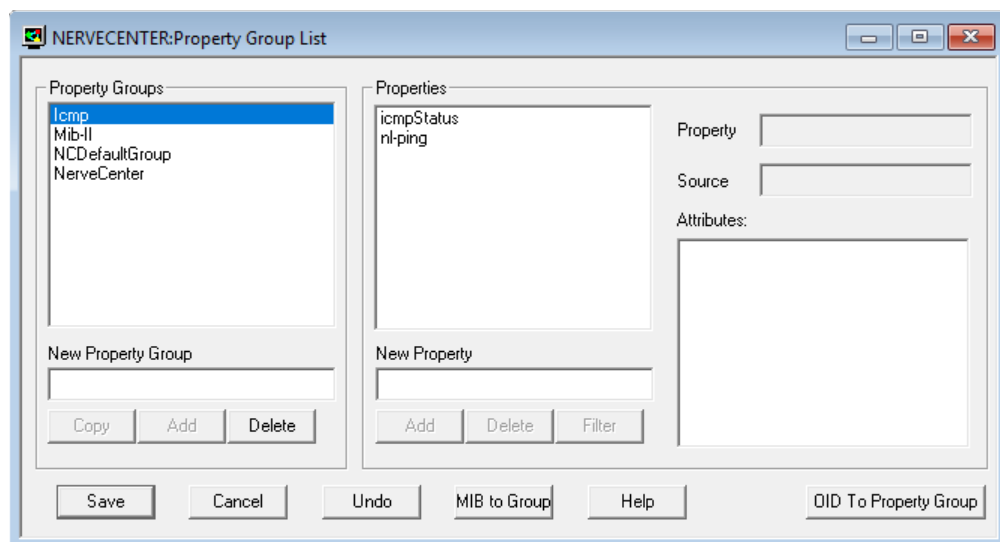
4. Select the **Copy** button, located below the New Property Group text field.
Your new property group appears in the Property Group list and is highlighted.
5. Use the procedure explained in the section [Creating a Property on page 132](#) to add one or more new properties to your property group.
6. Select the **Save** button.

Based on the Contents of MIBs

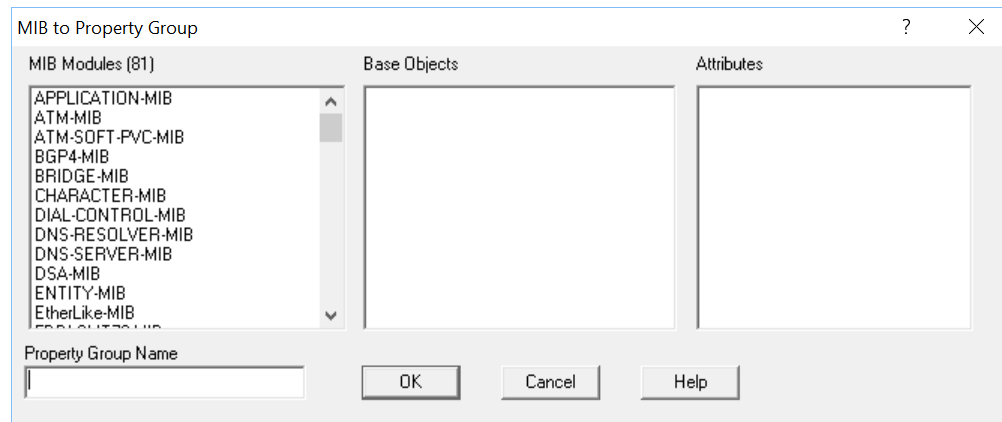
If you purchase a new device that comes with a new vendor MIB, your NerveCenter administrator should incorporate the new MIB into NerveCenter's compiled MIB so that you can take advantage of the new information provided by the vendor. In addition, you should create a new property group that contains properties for all the base objects in the new MIB. Why? Recall that a node's property group must contain properties for each of the MIB base objects you monitor on the node. If you want to poll the new device for the values of the attributes belonging to the new MIB objects, you need properties for the new base objects in the device's property group.

TO CREATE A NEW PROPERTY GROUP BASED ON THE CONTENTS OF ONE OR MORE MIBS

1. From the client's **Admin** menu, select **Property Group List**.
The Property Group List window is displayed.

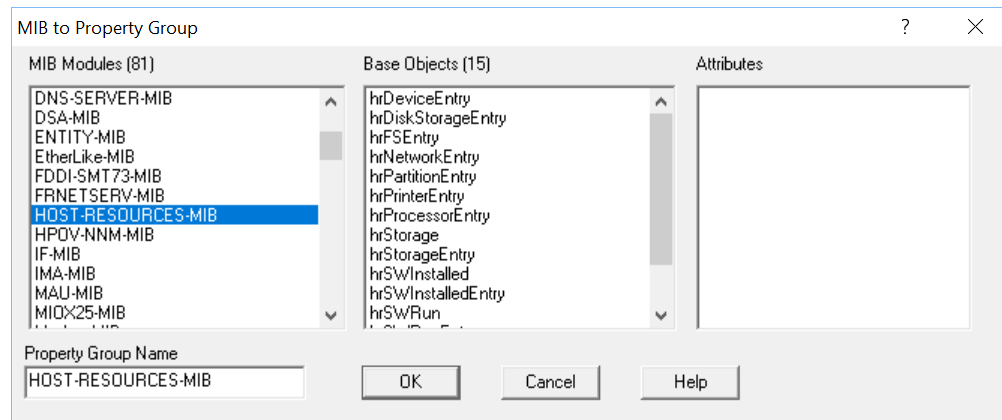


2. Select the **MIB to Group** button at the bottom of the window.
NerveCenter displays the MIB to Property Group window.



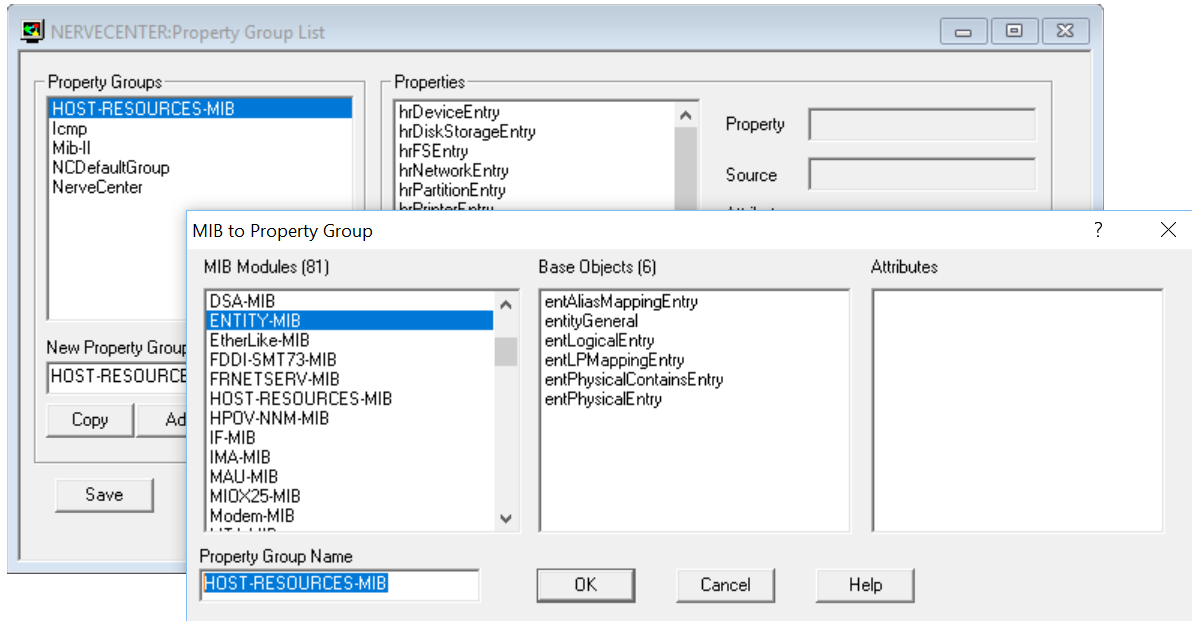
All of the MIBs in NerveCenter's compiled MIB are displayed in the MIB list. If you select one of the MIBs in the list, the names of the base objects for that MIB are displayed in the Base Objects list.

3. Select from the **MIB** list a MIB whose base objects you want to become properties in your new property group.
4. Enter a name for your property group in the **Property Group Name** text field. Or leave there the default name that NerveCenter has supplied.

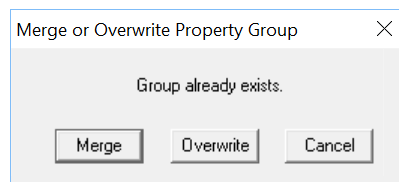


5. Select the **OK** button.
The MIB to Property Group window is dismissed, and the name of your new property group appears in the Property Group list in the Property Group List window. If you wanted to base your property group on just one MIB, you're finished. If you want the new property group to contain the names of the base objects from more than one MIB, continue with [Step 6](#).
6. In the Property Group List window, select the **MIB to Group** button again.
The MIB to Property Group window is displayed.

- In the MIB to Property Group window, select from the **MIB** list another MIB whose base objects you want included in your property group.



- Enter in the **Property Group Name** field the same name you used in Step 4.
 - Select the **OK** button.
- The Merge or Overwrite Property Group window is displayed.



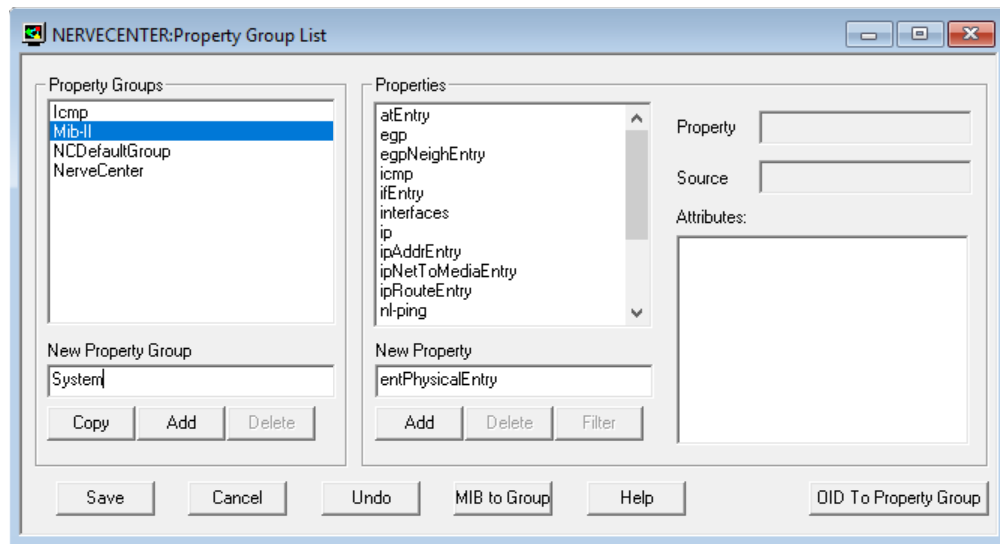
- Select the **Merge** button.
- Repeat Step 6 through Step 10 if necessary.
- Select the **Save** button.

Adding Properties Manually

If you need a property group that contains only a few properties—maybe a couple of base object names and one user-defined property—you can create an empty property group and then add properties to it by hand.

TO CREATE AN EMPTY PROPERTY GROUP AND THEN ADD PROPERTIES TO IT

1. From the client's **Admin** menu, select **Property Group List**.
The Property Group List window is displayed.
2. Type the name of your new property group in the **New Property Group** text field.



3. Select the **Add** button under the New Property Group text field.
Your new property group appears in the Property Group list and is highlighted. Note that no properties are listed in the Property list since the property group is empty.
4. To add one or more properties to the new property group, perform the steps covered in the section [Creating a Property on page 132](#).
5. Select the **Save** button.

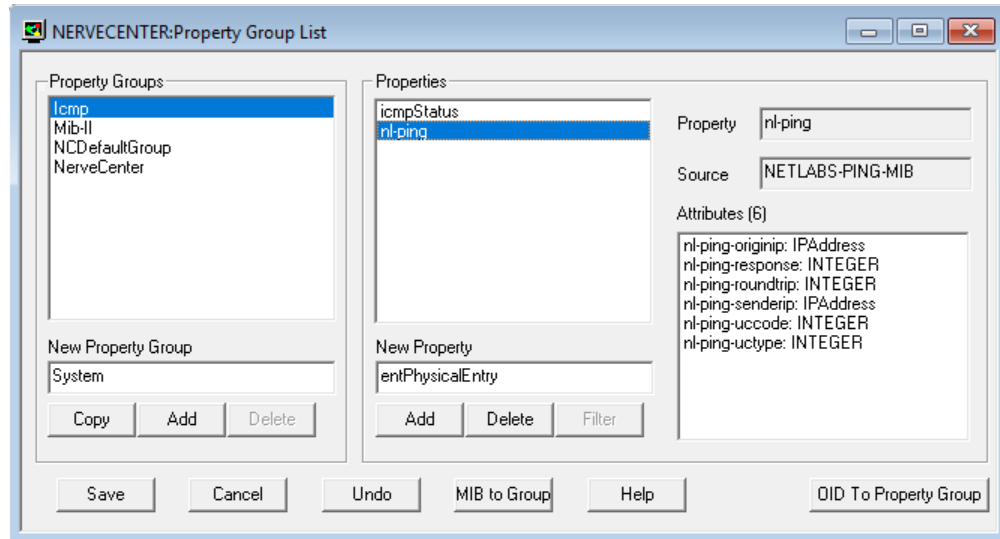
The nl-ping Property

NerveCenter includes a special “nl-ping” property that identifies that an operation utilizes ICMP instead of SNMP. In the context of polling, nl-ping can be used to indicate either of two types of behavior:

- To indicate that a poll should use an ICMP Echo (or Ping) request, not an SNMP operation.
- To view the details of an ICMP response to an SNMP operation. In the context of event monitoring, nl-ping is used within a Trap Mask to decode ICMP message details.

The nl-ping property is defined in NerveCenter's NETLABS-PING-MIB module (nl-ping.asn1). This can be pulled in with the **MIB to Group** button on the Property Group List (select **NETLABS-PING-MIB** in the left-hand MIB column and **nl-ping** will appear in the right-hand Base Objects column).

When using the default database included with NerveCenter, "nl-ping" is included in many of the Property Groups such as "Icmp", "Mib-II" and many of the vendor-based Property Groups. This provides ease of reference to this ability to access ICMP-related information.



You can find more information in the following sections:

- Use of nl-ping in polling and in poll conditions is presented in [The Basic Procedure for Creating a Poll Condition on page 167](#).
- Use of nl-ping in Poll response handling is presented in [ICMP Responses on page 221](#).
- Use of nl-ping in Trap Notification handling is presented in [How NerveCenter Decodes ICMP Events on page 195](#).

Filtering Properties

When polling, NerveCenter steps through every property value to see if it meets the correct conditions to advance to the next state. Some properties defined by MIBs, such as ifEntry, can consist of hundreds of entries. In these cases, a single poll can use valuable resources as NerveCenter steps through every value, looking for the value that meets the condition defined in the poll. If the property has an associated MIB table index, and you know the specific property value or range of values you want to monitor, you can use property filters to constrain your poll.

[Table 13](#) describes the property filter values.

Table 13: Property Filter Attributes

Attribute	Value
Property Index	<p>Automatically populated by NerveCenter.</p> <p>The MIB attribute that acts as the index of the property's associated MIB table. This MIB attribute can only consist of a single attribute; property filters do not support multiple field indices.</p>
Datatype	<p>Automatically populated by NerveCenter.</p> <p>The datatype of the Property Index attribute.</p> <p>Property filters work with the following datatypes: IpAddress, Integer32, INTEGER, Counter, Gauge, TimeTicks, Counter32, UInteger32, and Counter64.</p> <p>The following datatypes are <i>not</i> supported: NULL, DisplayString, Bits, NetworkAddress, OctetString, ObjectID, Timeout, and TimeStamp.</p>
Default Range	<p>Automatically populated by NerveCenter.</p> <p>The allowed values of the Property Index attribute as specified in the MIB.</p>
Filter Index Range	<p>The Property Index values you want to include. The Filter Range can consist of the following:</p> <ul style="list-style-type: none"> ■ a comma delimited list of individual values For example: 7,12,19,34 ■ a dash (-) delimited pair of values to indicate the lower and upper bounds of a range For example: 18-35 ■ a combination of these values For example: 2,4,35-38 <p>Property filters have the following limitations:</p> <ul style="list-style-type: none"> ■ the maximum value is 2,147,483,647 ■ no negative numbers ■ the filter range can be no more than 200 characters ■ the filter can contain no more than 1,000 unique values. For example, 2000-4000 is not a valid filter index range because it encompasses more than 1,000 values.

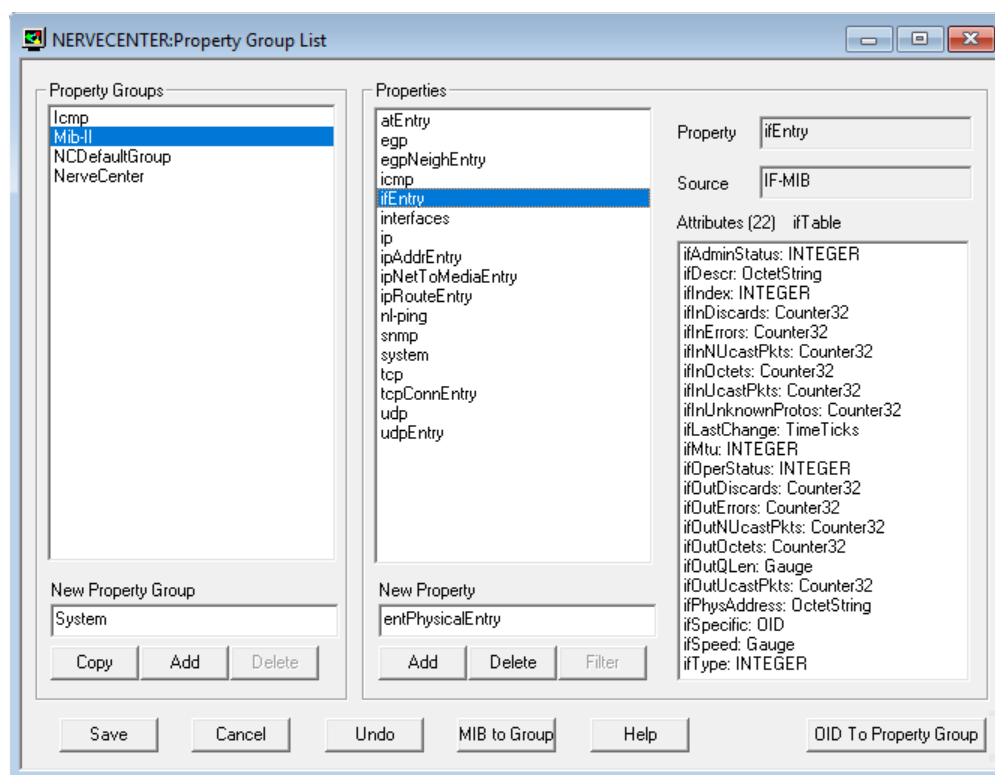
Note: Property filters are only applied to the property belonging to the specified property group. The same property in a different property group will not be filtered.

Caution: Before creating or editing Property Filters, you must turn off all alarms using the property group you are editing.

TO CREATE A PROPERTY FILTER

1. From the client's **Admin** menu, select **Property Group List**.

The Property Group List window is displayed.



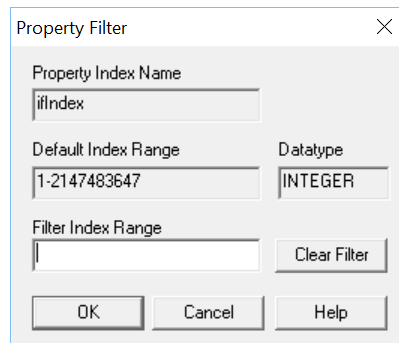
2. Select a **Property Group**.

3. Select a **Property**.

If the property has an associated MIB table index that meets the requirements described in [Table 13](#), the Filter button is active. You must save a new property group or new property before the Filter button activates.

4. Click **Filter**.

The Property Filter window is displayed.



5. Enter a **Filter Index Range**.

See [Filter Index Range](#) on [page 140](#) for acceptable values. Click **Clear Filter** to delete the Filter Index Range.

6. Click **OK**.

You return to the Property Group List dialog box.

7. Click **Save**.

The filter is not saved until you save the property group. If you click Cancel in the Property Group List, all filters will be lost.

Note: Filters are applied when polled by an alarm with a SubObject or Instance scope only.

TO EDIT A PROPERTY FILTER

1. From the client's **Admin** menu, select **Property Group List**.
The Property Group List window is displayed.
2. Select the **Property Group** with the property filter.
3. Select the **Property** with the property filter.
4. Click **Filter**.
The Property Filter window is displayed, with the existing filter information.
5. Edit the **Filter Index Range**.
See [Filter Index Range on page 140](#) for acceptable values. Click **Clear Filter** to delete the Filter Index Range.
6. Click **OK**.
You return to the Property Group List dialog box.
7. Click **Save**.

Assigning a Property Group to a Node

When a node is created, it is assigned a property group, and this property group determines which behavior models NerveCenter uses to manage the node. Of course, this property group assignment isn't permanent. You can change the assignment manually, or a behavior model being used to manage the node can change it.

This section discusses a number of ways in which you can assign a property group to a node and explains when you would use each method. For further information, see the following subsections.

- [Using the Node Definition Window on the next page](#)
- [Using the Node List Window on page 145](#)
- [Using the AssignPropertyGroup\(\) Function on page 147](#)
- [Using the Set Attribute Alarm Action on page 155](#)
- [Using OID to Property Group Mappings on page 158](#)

Using the Node Definition Window

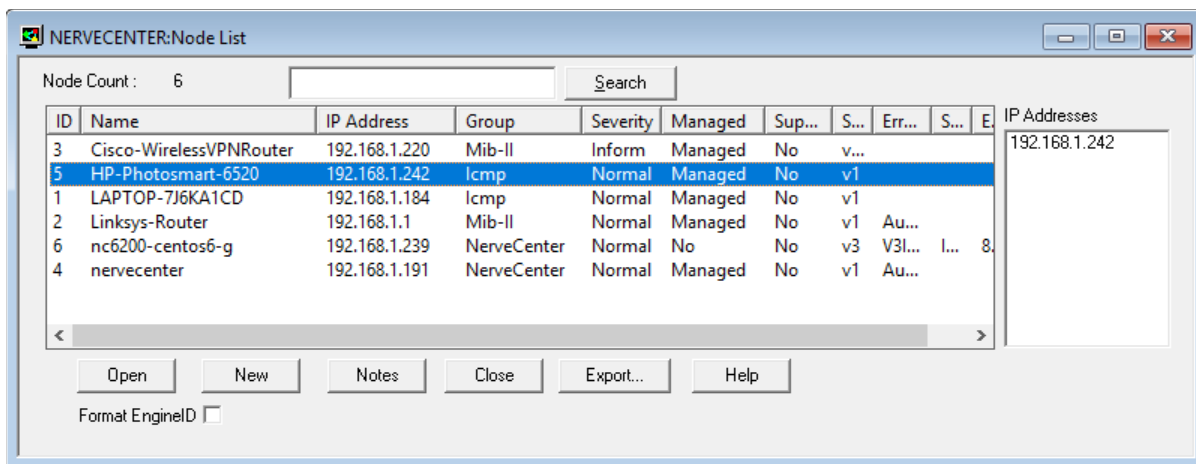
One way to change the property group of a node is to open the Node Definition window for that node and to change the value of the Group field. This method is an appropriate way to change a node's property group if:

- You know in advance which node or nodes need the new property group
- Only one node or a few nodes need the change

TO CHANGE A NODE'S PROPERTY GROUP USING THE NODE DEFINITION WINDOW

1. From the client's **Admin** menu, choose **Node List**.

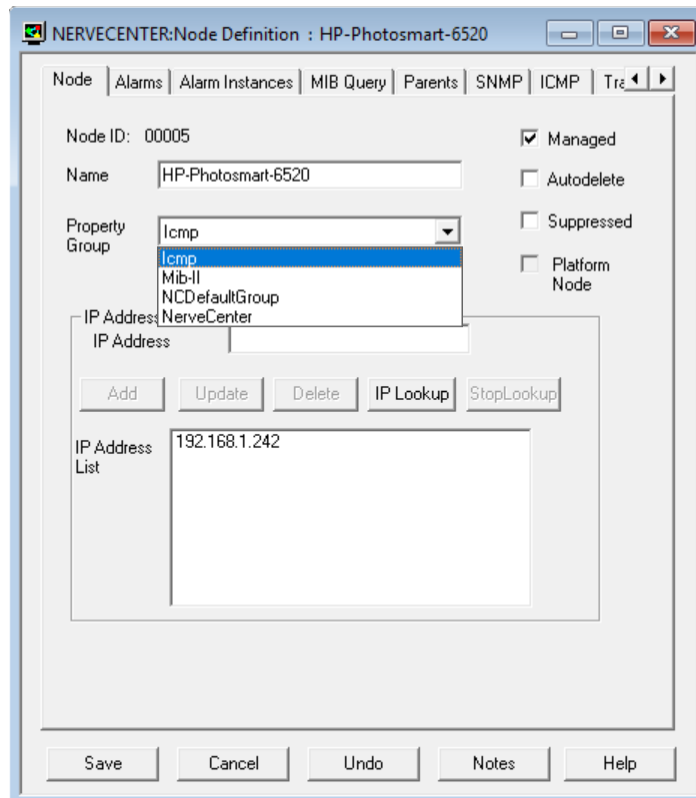
NerveCenter displays the Node List window.



2. Highlight the name of the node whose property group you want to change.

3. Select the **Open** button.

The Node Definition window appears. This window enables you to edit the properties of the node you selected.



4. Select a new property group from the **Group** drop-down list.
5. Select the **Save** button.

Repeat this procedure for any additional nodes you want to assign to a new property group.

Using the Node List Window

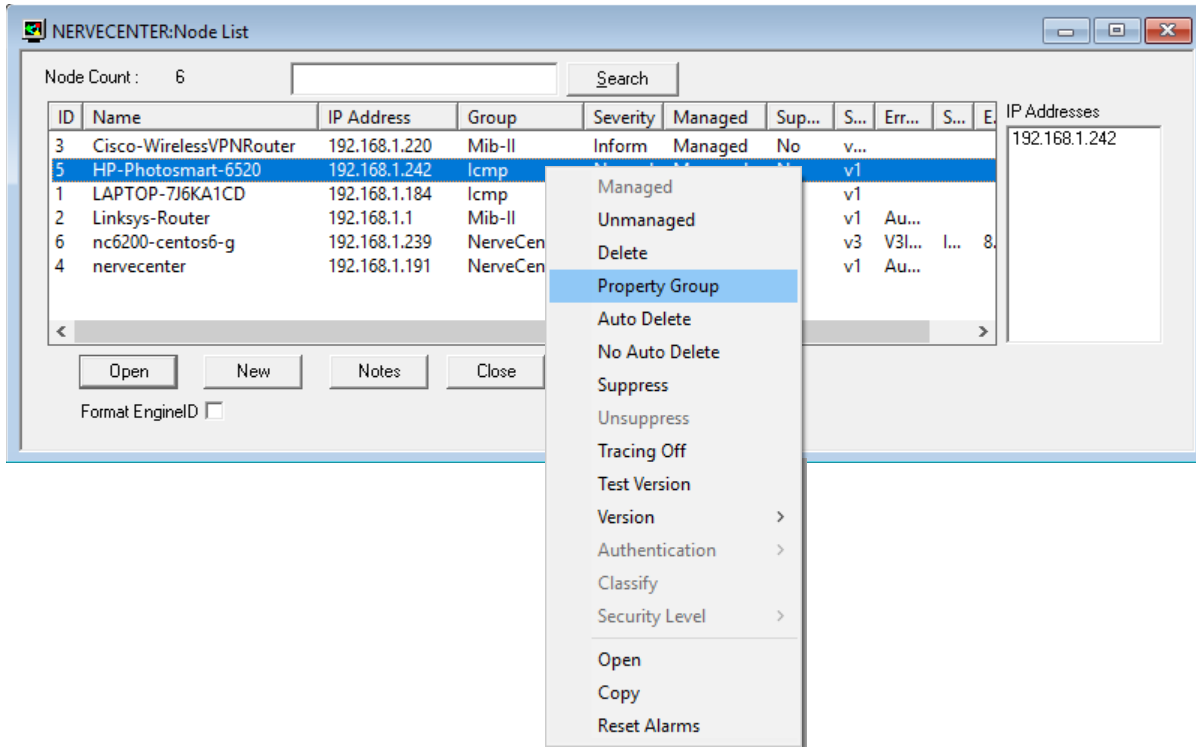
You can change the property group of a set of nodes from the Node List window, using a popup menu accessible from that window. It is appropriate to use this method of property group assignment if:

- You need to change the property group for more than a couple of nodes
- You want to assign the same property group to each of the nodes
- You know in advance which nodes you want to modify

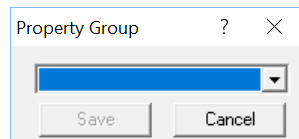
TO CHANGE THE PROPERTY GROUP FOR A SET OF NODES FROM THE NODE LIST WINDOW

1. From the client's **Admin** menu, choose **Node List**.

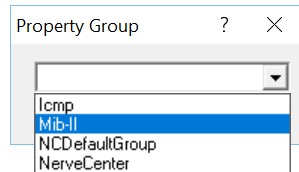
NerveCenter displays the Node List window.



2. Select one node whose property group you want to change. Then hold down the Ctrl key and select the remainder of the nodes you want to modify.
3. With your cursor positioned over one of the highlighted entries, press the right mouse button to bring up the node-management popup menu, and select **Property Group** from the menu. NerveCenter displays the Property Group Edit dialog box.



4. Select a property group from the drop-down list.



5. Select the **Save** button.

Using the AssignPropertyGroup() Function

In addition to being able to assign property groups to nodes manually using the NerveCenter user interface, you can use the `AssignPropertyGroup()` function in a behavior model to change a node's property group dynamically. This function can appear in a poll condition, a trap mask trigger function, or a Perl subroutine.

The syntax for this function is shown below:

```
AssignPropertyGroup ("PropertyGroupName")
```

The property group whose name is passed to the function must already exist.

For further information about how to use this function in a poll condition, a trigger function, or a Perl subroutine—and for information on when it's appropriate to use the function in each of these contexts—see the sections listed below:

- [In a Poll Condition below](#)
- [In a Trigger Function on page 149](#)
- [In a Perl Subroutine on page 151](#)

In a Poll Condition

Suppose you want to change the property group assignment for all of your Cisco routers in Building 6. You can collect the names or IP addresses of all these nodes and change their property groups manually using the NerveCenter user interface. However, this can be an error prone process. All you have is your list of routers to make sure that you assign the new property group to exactly the right set of nodes.

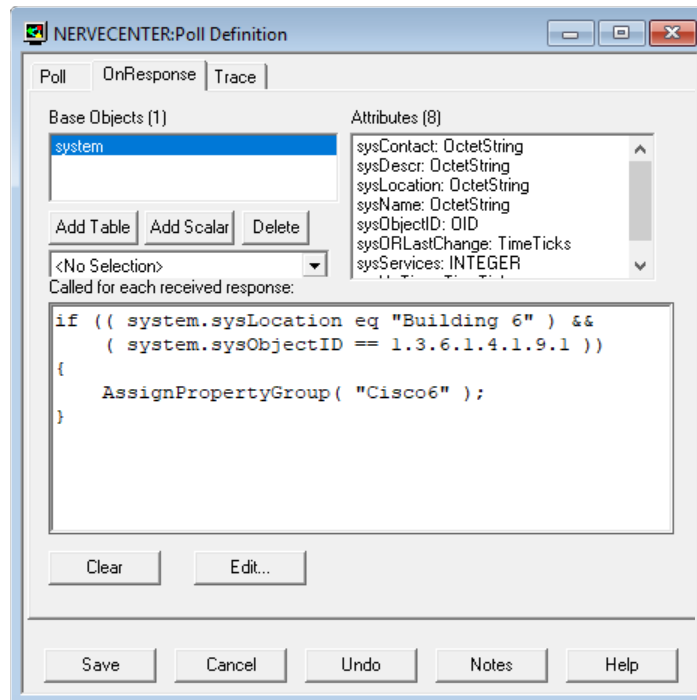
Alternatively, you can create a poll that will detect whether a polled node is a Cisco router located in Building 6 and will assign the new property group only to nodes that meet these criteria.

Note: The instructions below are not intended to explain in detail how to create this type of poll. Creating polls is a fairly large topic and is covered in [Using Polls on page 161](#). These instructions cover only the general procedure for incorporating a call to `AssignPropertyGroup()` into a poll condition.

This procedure details how to define a poll condition that changes the property group of each Cisco router in Building 6:

TO DEFINE A POLL CONDITION THAT CHANGES THE PROPERTY GROUP (SAMPLE)

1. Display the **Poll Condition** page in the **Poll Definition** window.



2. Create the condition that determines whether you want to call `AssignPropertyGroup()`:

```
if ((system.sysLocation eq "Building 6") &&
    (system.sysObjectID == 1.3.6.1.4.1.9.1))
```

3. Add a block including a call to `AssignPropertyGroup()` to the preceding condition:

```
if ((system.sysLocation eq "Building 6") &&
    (system.sysObjectID == 1.3.6.1.4.1.9.1)) {
    AssignPropertyGroup("Cisco6");
}
```

This example assumes that the new property group is named `Cisco6`.

Note: Your poll condition must also include a call to `FireTrigger()`; otherwise, you won't be able to save the poll.

4. Select the **Save** button to save your poll.

Before NerveCenter will use this poll, there must be an enabled alarm in which the poll can cause a state transition.

Caution: When a poll changes a node's property group, any alarm instances that have been created for that node are deleted.

In a Trigger Function

Here's a simple example of when you might use the `AssignPropertyGroup()` function in a trap mask trigger function. Suppose that you want to use NerveCenter's Authentication behavior model to monitor your network for excessive SNMP authentication failures. This model includes a trap mask and two polls and looks for three authentication failures on a single node within ten minutes.

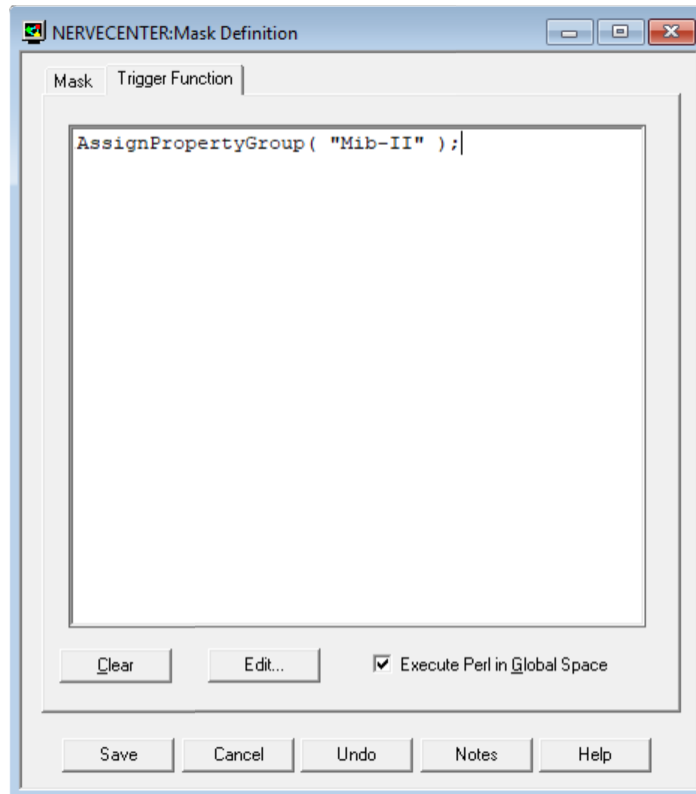
You could enable the behavior model by assigning to the nodes you want to monitor a property group that contains the property `snmp` and turning on the Authentication alarm. But let's say that you don't want to monitor nodes that have never experienced an authentication failure, because the model does involve some polling. To monitor only nodes whose agents have sent authentication failure traps, you can initially assign your nodes a property group that doesn't contain the property `snmp`. You can then define a trap mask that looks for authentication failure traps and changes the property group of the nodes from which it receives these traps. Let's assume that the new property group is called `Mib-II` and contains the property `snmp`.

Note: The instructions below are not intended to explain in detail how to create this type of trap mask. Creating masks is a fairly large topic and is covered in "[Using Trap Masks](#)" on page 193. These instructions cover only the general procedure for incorporating a call to `AssignPropertyGroup()` into a trigger function.

TO DEFINE A TRAP MASK

This procedure defines a trap mask that changes the property group of each node that issues an authentication failure trap, you would:

1. Create a trap mask that looks for a generic trap 4.
2. Indicate that the trap mask will use a trigger function instead of a simple trigger.
3. Display the **Trigger Function** page in the Mask Definition window.



4. Type in your call to `AssignPropertyGroup()`:

```
AssignPropertyGroup("Mib-II");
```

You can make this property-group assignment conditional, based on the value of a variable binding if you need to. In the present case, such a condition isn't necessary.

5. Also type in a call to `FireTrigger()`;

```
FireTrigger("TriggerName");
```

Remember that before NerveCenter will use this mask, there must be an enabled alarm in which the mask can cause a state transition.

6. Save your trap mask.

Caution: When a mask changes a node's property group, any alarm instances that have been created for that node are deleted.

In a Perl Subroutine

Another place from which you can call the `AssignPropertyGroup()` function is a Perl Subroutine alarm-transition action. This is the appropriate context for using this function if you want to perform your property-group assignment conditionally, based on information that is available from with a Perl subroutine, but not elsewhere. For example, a Perl subroutine associated with an alarm transition has access to the name of the property group of the node that triggered the transition. You could use this information to change a node's property group only if:

- An alarm transition containing the appropriate Perl Subroutine action is caused by a trigger associated with the node
- The node currently has a particular property group

For a complete list of the information that is available to a Perl subroutine, see the section [NerveCenter Variables on page 301](#).

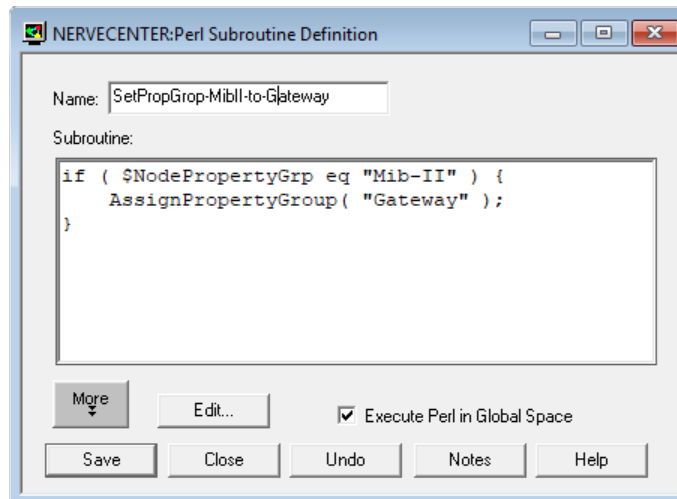
Note: The instructions below do not explain in detail how to create a Perl subroutine or how to create an entire alarm. They explain only how to add to an alarm transition a Perl Subroutine action that will change the property group of a node. For complete information about creating Perl subroutines, see the section [Perl Subroutine on page 294](#), and for complete information about creating alarms, see [Using Alarms on page 237](#)

TO ADD A PERL SUBROUTINE TO AN ALARM TRANSITION

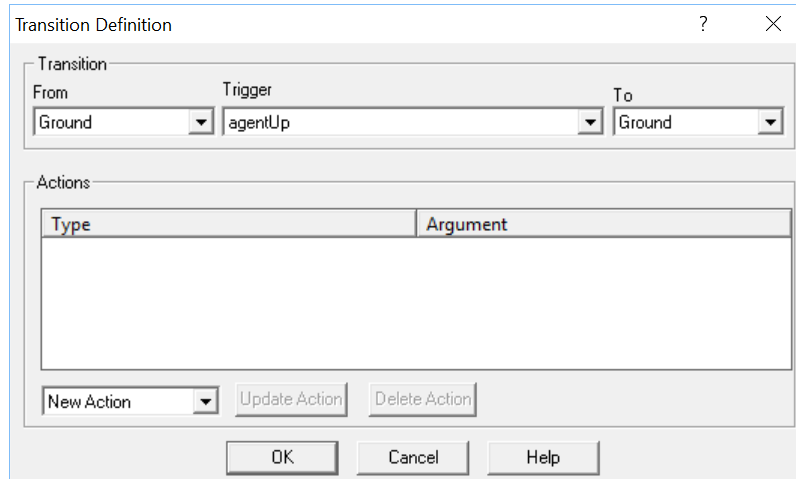
The procedure below explains how to add to an alarm transition a Perl Subroutine action that assigns the property group Gateway to the node associated with the trigger that caused the transition. The property group is assigned only if the node's current property group is Mib-II.

1. Use the Perl Subroutine Definition window to create your Perl subroutine.

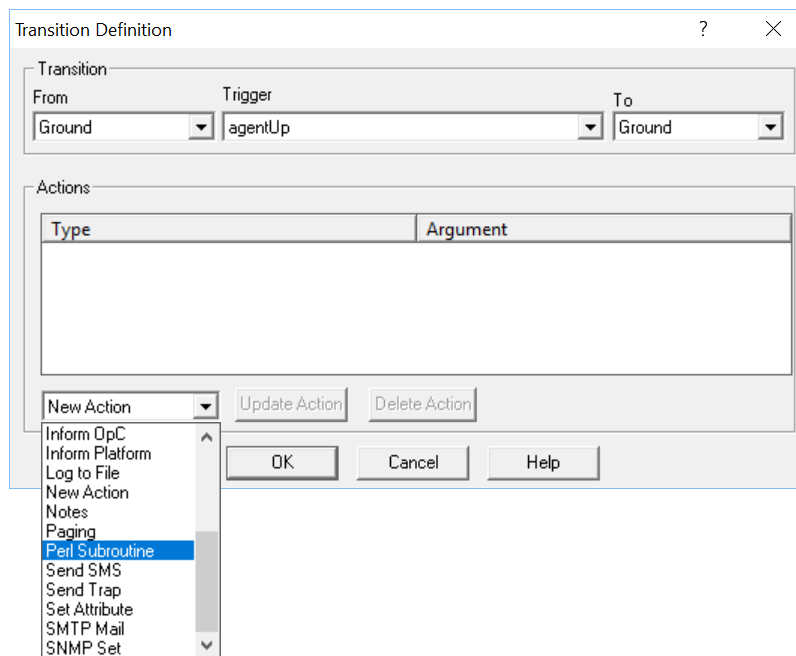
The subroutine should look something like this:



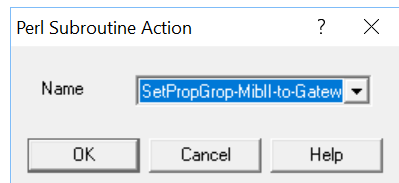
2. In the Alarm Definition window, open the **Transition Definition** dialog by double-clicking on the transition to which you want to add the Perl Subroutine action.



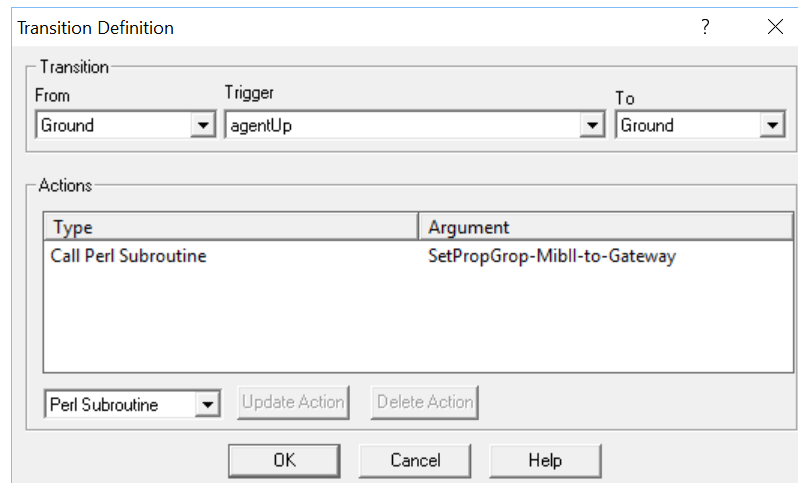
3. Select the **New Action** list.
A list of available actions is displayed.



- Select the **Perl Subroutine** action.
NerveCenter displays the Perl Subroutine Action dialog box.



- Select the name of the subroutine you created in [Step 1](#) from the **Name** list box.
- Select the **OK** button in the Perl Subroutine Action dialog.



The dialog is dismissed, and the newly defined action appears in the Actions list in the Transition Definition dialog.

- Select the **OK** button in the Transition Definition dialog.
- Select the **Save** button in the Alarm Definition window.

Caution: When a Perl subroutine changes a node's property group, any alarm instances that have been created for that node are deleted.

Using the Set Attribute Alarm Action

There are two ways to change a node's property group using alarm-transition actions: using the Perl Subroutine action and using the Set Attribute action. For information on changing a node's property group using the Perl Subroutine action, see the section [In a Perl Subroutine on page 151](#). Using a Perl Subroutine action to change a property group is appropriate when you want to use Perl to do something more complex than simply change the property group of the node associated with the trigger that causes the alarm transition (or the property group of any other node, for that matter). If the only action you would take from a Perl subroutine is to change a property group, you should use the Set Attribute action instead. This approach will save you the trouble of having to write and compile a Perl subroutine.

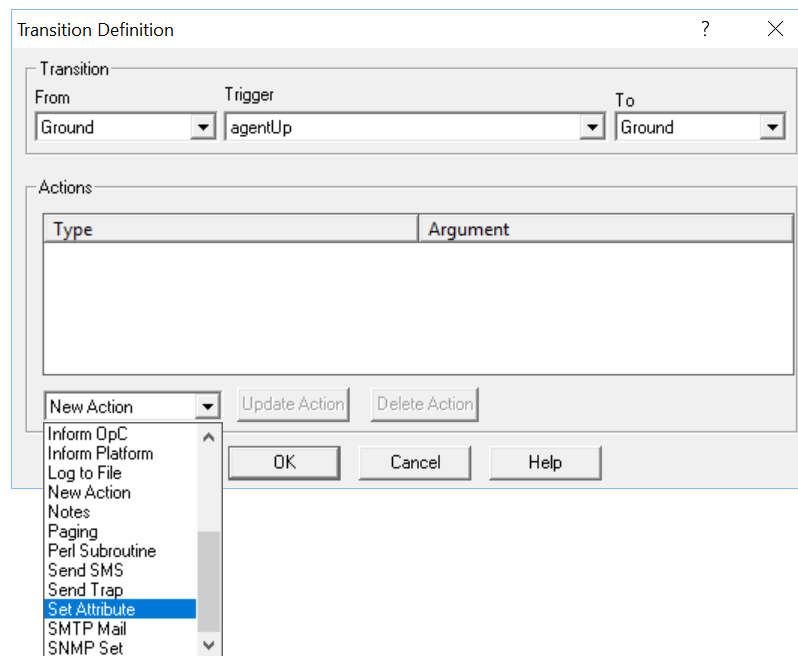
Note: The instructions below do not explain how to create an entire alarm. They explain only how to add to an alarm transition a Set Attribute action that will change the property group of a node. For complete information about creating alarms, see ["Using Alarms" on page 237](#).

TO ADD TO AN ALARM TRANSITION A SET ATTRIBUTE ACTION THAT CHANGES A NODE'S PROPERTY GROUP

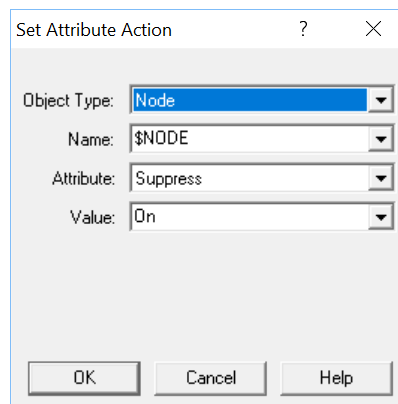
1. Open the **Transition Definition** dialog by double-clicking on the transition to which you want to add the Set Attribute action.

The screenshot shows the 'Transition Definition' dialog box. The 'Transition' section has three dropdown menus: 'From' set to 'Ground', 'Trigger' set to 'agentUp', and 'To' set to 'Ground'. The 'Actions' section contains a table with two columns: 'Type' and 'Argument'. Below the table are three buttons: 'New Action', 'Update Action', and 'Delete Action'. At the bottom of the dialog are three buttons: 'OK', 'Cancel', and 'Help'.

2. Select the **New Action** list.
A list of available actions is displayed.

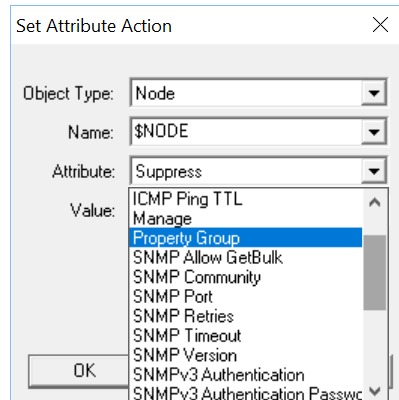


3. Select the **Set Attribute** action.
NerveCenter displays the Set Attribute Action dialog.



4. Leave the **Object Type** value set to **Node** since you want to set an attribute of a node.
5. Usually you'll leave the **Name** value set to **\$NODE**.
\$NODE stands for the name of the node associated with the trigger that caused the alarm transition. However, you can change the value to the name of any node in the NerveCenter database if you know in advance the name of the node whose property group you want to change.

6. Select **Property Group** from the **Attribute** drop-down list.



Set Attribute Action

Object Type: Node

Name: \$NODE

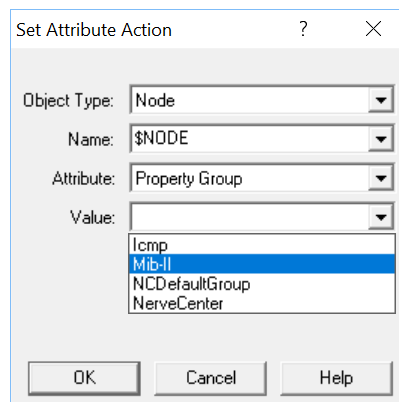
Attribute: Suppress

Value: ICMP Ping TTL
Manage
Property Group
SNMP Allow GetBulk
SNMP Community
SNMP Port
SNMP Retries
SNMP Timeout
SNMP Version
SNMPv3 Authentication
SNMPv3 Authentication Passw

OK

7. Select a property-group name using the **Value** drop-down list.

The property group you choose will become the new property group for the node you chose in [Step 5](#) whenever this alarm transition takes place.



Set Attribute Action

Object Type: Node

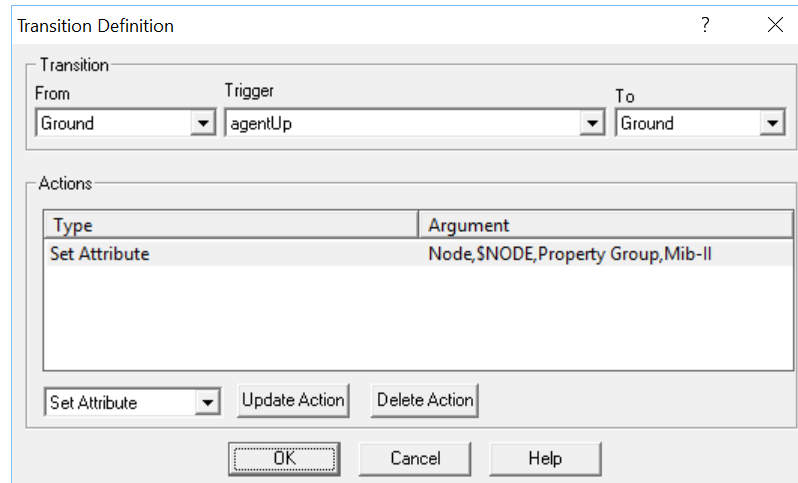
Name: \$NODE

Attribute: Property Group

Value: Icmp
Mib-II
NCDefaultGroup
NerveCenter

OK Cancel Help

8. Select the **OK** button in the Set Attribute Action dialog.
The dialog is dismissed, and the newly defined action appears in the Actions list in the Transition Definition dialog.



9. Select the **OK** button in the Transition Definition dialog.
10. Select the **Save** button in the Alarm Definition window.

Caution: When a Set Attribute alarm action changes a node's property group, any alarm instances that have been created for that node are deleted.

Using OID to Property Group Mappings

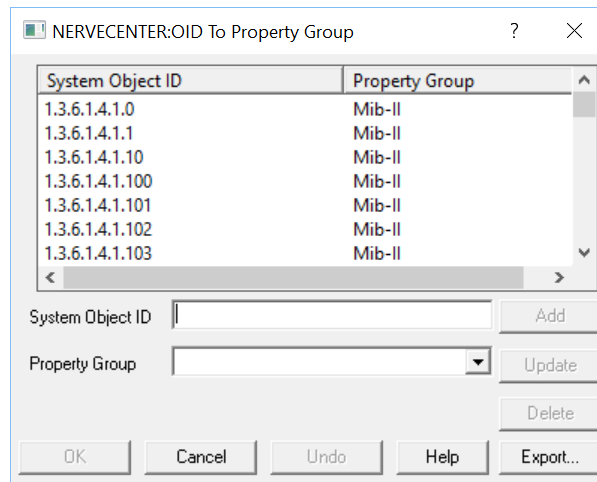
When a node is first written to the NerveCenter database, it is assigned a property group based on the object ID of the node. For example, a Cisco router with an OID of 1.3.6.1.4.1.9.1 is, by default, assigned a property group of CISCO-ROUTER-9.x-MIB. The assignments are based on a table of mappings between OIDs and property groups. If no mapping exists for a particular device, that device is assigned the default property group NCDefaultGroup.

Using the NerveCenter client, you can add entries to, or change entries in, this OID-to-property-group table. The new mappings will affect any nodes that are added to the NerveCenter database after you make your changes.

TO ADD A NEW OID-TO-PROPERTY-GROUP MAPPING

1. From the client's **Admin** menu, choose **OID to Group**.

The OID to Property Group dialog is displayed.



2. Enter an object identifier in the **System Object** text field.
3. Enter the name of a property group in the **Property Group** text field.
4. Select the **Add** button.
5. Select the **Save** button.

Tips for Using Property Groups and Properties

Using property groups and properties is mainly a matter of common sense; however, the sections below give you a few suggestions for using them effectively.

Categorizing Nodes

We've said that property groups enable you to create groups of nodes, each of which is managed by a set of behavior models. As you create your groups, it's helpful to list a variety of criteria for categorizing your nodes and then to use the criteria that make the most sense for your network. For example, some criteria you could use in classifying your nodes are:

- Type of device (workstation, server, router)
- Location
- Importance (Which nodes need to be managed most closely?)
- Supported MIBs
- Business function

Apply whatever set of criteria is appropriate for your site.

Move from the General to the Specific

Set up property groups that establish general groups of devices first. Then create subcategories of nodes as necessary.

For instance, suppose that you have MIB-II agents running on all of your computers, including servers. You want to monitor the servers more closely than the personal computers, so you copy the existing Mib-II property group, name the copy *Server*, and add to the copy the property *server*. You can now set up polls and alarms that take one action, such as sending an e-mail message, when any workstation is unreachable, and another action, such as paging an administrator, when a server is unreachable.

Or maybe you want to refine how you monitor servers so that you can distinguish file servers from print servers. You can set up two new property groups, each a copy of *Server*. Name one *Fserver* and add the property *fserver*, and name the other *Pserver* and add the property *pserver*. Note that both groups still contain the property *server* because each is a copy of the *Server* property group. You can then set up polls and alarms to perform one action when any server is unreachable, perform a different action when a file server is unreachable, and perform a third action when a print server is unreachable.

MIB Objects

The property group for a device should contain a property for every MIB base object that might be used in a poll condition by a poll designed to contact that node. For further information on building poll conditions, see [Writing a Poll Condition on page 166](#).

If a base object is not in the node's property group, polls whose poll conditions refer to that object will not contact the node.

NerveCenter polls enable you to retrieve information from SNMP agents on devices to determine the status of those devices. Figure 25 depicts the role that a poll plays in a behavior model

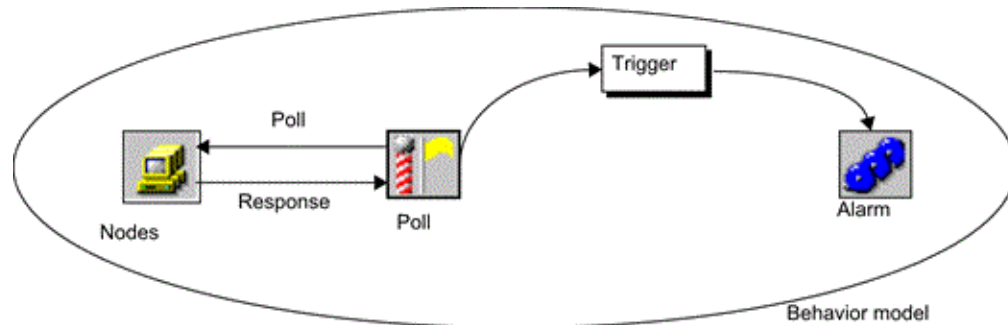


Figure 25: The Role of a Poll in a Behavior Model

To function as part of a behavior model, a poll must be tied to one or more alarms by means of one or more triggers. If the poll does not define a trigger that can affect a pending alarm transition, the poll is never sent to a device. This behavior is part of NerveCenter’s smart polling feature.

Other aspects of this smart polling feature are that NerveCenter doesn’t send a poll to a node unless the poll’s property is in the node’s property group and that NerveCenter never sends a suppressible poll to a suppressed node. Together, these behaviors sharply curtail the amount of network traffic NerveCenter generates by polling SNMP agents.

The remainder of this chapter explains in detail how to create and work with polls.

Listing Polls

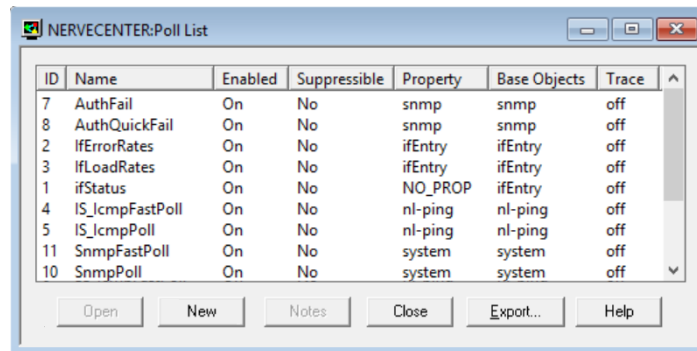
This section explains how to display a list of the polls currently defined in the NerveCenter database. The section also explains how to view the definition of a particular poll.

For information on creating a new poll, see [Defining a Poll on page 164](#).

TO DISPLAY A LIST OF POLLS AND THEN DISPLAY A PARTICULAR POLL'S DEFINITION

1. From the client's **Admin** menu, choose **Poll List**.

NerveCenter displays the Poll List window.



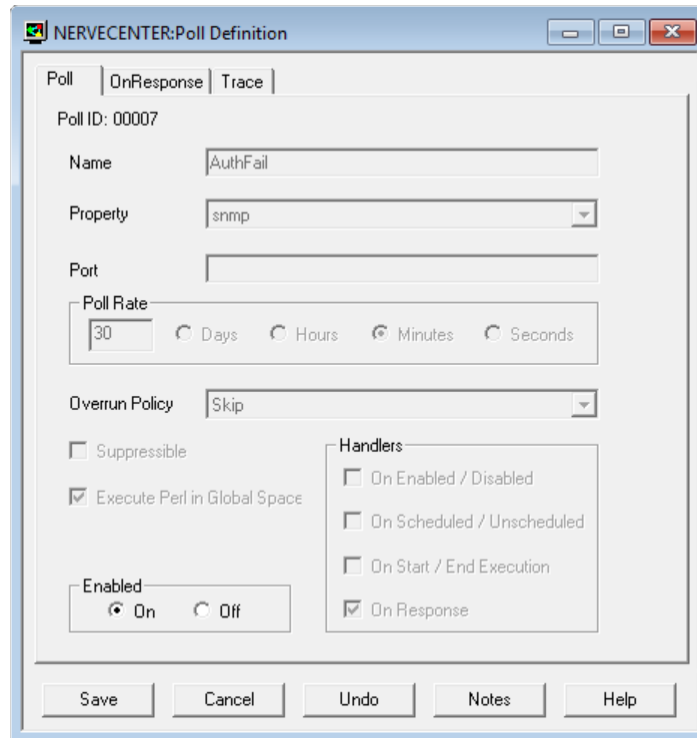
ID	Name	Enabled	Suppressible	Property	Base Objects	Trace
7	AuthFail	On	No	snmp	snmp	off
8	AuthQuickFail	On	No	snmp	snmp	off
2	IfErrorRates	On	No	ifEntry	ifEntry	off
3	IfLoadRates	On	No	ifEntry	ifEntry	off
1	ifStatus	On	No	NO_PROP	ifEntry	off
4	IS_IcmpFastPoll	On	No	nl-ping	nl-ping	off
5	IS_IcmpPoll	On	No	nl-ping	nl-ping	off
11	SnmpFastPoll	On	No	system	system	off
10	SnmpPoll	On	No	system	system	off

Buttons: Open, New, Notes, Close, Export..., Help

This window lists all NerveCenter polls and provides a brief definition of each. For each poll, the window specifies a name and the following information:

- Whether the poll is currently enabled
 - Whether the poll is suppressible
 - The poll's property
 - The name of the base object used to build the poll condition
2. Select a poll from the poll list.

3. Select the **Open** button
NerveCenter displays the Poll Definition window.



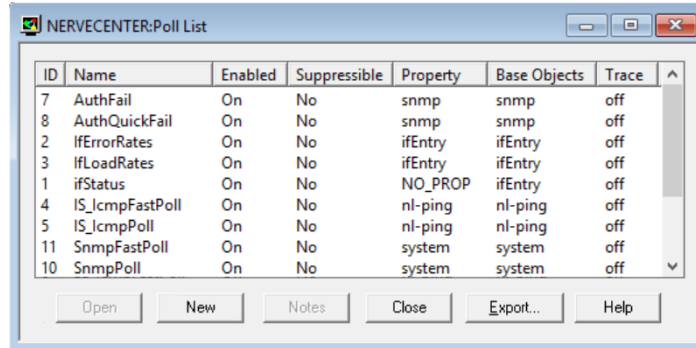
The poll defined in this figure is named AuthFail. Every thirty minutes, the poll is sent to nodes whose property group includes the property snmp, and the poll checks for an increase in the value of snmpInBadCommunityNames or snmpInBadCommunityUses. If the poll finds an increase in either of these values, it fires the trigger authFail; otherwise, it does not fire a trigger. The poll is suppressible and is currently not enabled. It must be enabled before NerveCenter will use its definition to poll any devices.

Defining a Poll

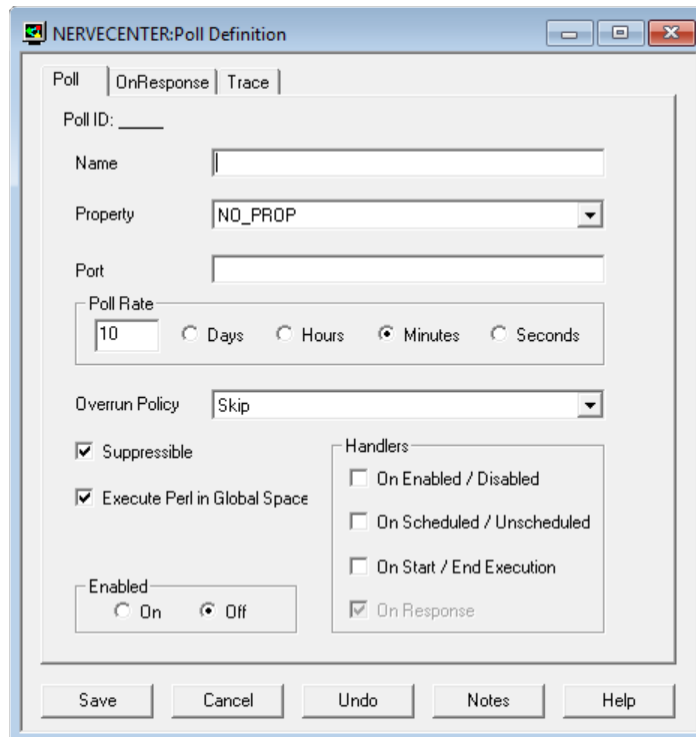
This section explains the steps required to create a new poll.

TO DEFINE A NEW POLL

1. From the client's **Admin** menu, choose **Poll List**.
NerveCenter displays the Poll List window.



2. Select the **New** button.
The Poll Definition window is displayed.



3. Make sure that the **Off** radio button is selected in the **Enabled** frame.
The poll must remain off until you've completed defining the poll and saved your definition. You must then turn the poll on for it to become part of a functioning behavior model.
4. In the Name text field, type a unique name for the poll.

Note: The maximum length for poll names is 255 characters.

5. From the Property list box, select a property, or leave the **Property** set to NO_PROP.
The property you choose limits which nodes NerveCenter can retrieve data from using this poll definition. The poll will contact only those nodes whose property group contains this property. (Note that the property can be a member of multiple property groups.)

If you don't want to restrict the poll to any subset of nodes, leave the field set at NO_PROP. The poll will target all managed nodes.
6. Usually, you'll leave the Port text field blank. However, if you want this poll to communicate with nodes on a port other than that specified in the nodes' definitions, enter that port number here.
7. Define the poll rate by entering a number in the Poll Rate text field and selecting either the **Hours**, **Minutes**, or **Seconds** radio button.

Note: When defining the poll rate, the interval should be equal to or greater than $(\text{numberOfRetries} + 1) * \text{retryInterval}$. Otherwise, NerveCenter can issue a second poll before the first one times out. The number of retries and the retry interval are defined on the SNMP tab in the NerveCenter Administrator.

Caution: Choosing a frequent poll rate can have a serious impact on network traffic, especially if the poll applies to numerous nodes.

8. Uncheck the Suppressible checkbox if you want to send this poll to a node even when the node is suppressed.
A suppressible poll does not poll a node whose state is suppressed. This feature prevents repeated polling of devices that are not capable of responding. The default value for a poll is suppressible.

There might be specific polls that you want to send to a node even when it is suppressed. For example, if you want to check on the status of a suppressed node to determine whether it has returned to normal, use an insuppressible poll.
9. Select the Poll Condition tab to display the Poll Condition page, and enter your poll condition. For details on how to construct this poll condition, see [Writing a Poll Condition on the next page](#).
10. Select the **Save** button to save your poll.
11. If you want to enable your poll now, set the poll's Enabled status to **On**, and then select the **Save** button again.

Writing a Poll Condition

Every poll must include a poll condition. This poll condition, which you write using Perl, specifies which MIB variables the poll should read, what conditions the values of those variables must meet, and what triggers will be fired each time a value makes a condition true. For example, the following poll condition detects whether a node's desired and current operational status are both up and, if they are, fires the trigger ifUp:

```
if (ifEntry.ifAdminStatus == up and ifEntry.ifOperStatus == up) {
    FireTrigger("ifUp");
}
```

Note that both the MIB variables referred to in this condition are children of the same base object (ifEntry). In a single poll condition, you can only refer to one base object. If the condition that you want to detect requires that you inquire about variables associated with multiple base objects, you must design multiple polls.

Another important point about poll conditions is that if a poll causes a trigger to be fired, that trigger's variable bindings will include a name-value pair for each MIB variable referred to in the poll condition and read by the poll. If such a trigger causes a logging action, the value of each variable used in the poll condition is written to the log.

Most poll conditions are very similar in structure. They follow this pattern:

```
if (condition1) {
    FireTrigger(arguments);
}
elsif (condition2) {
    FireTrigger(arguments);
}
else {
    FireTrigger(arguments);
}
```

The conditions can be arbitrarily complex, and the FireTrigger() function fires a trigger, whose name, subobject, and node you can control.

Note: The maximum length for trigger names is 255 characters.

Because a poll condition is written in Perl, you can use any data types, operators, and functions that Perl understands in this condition. Also, you can make use of a number of functions and one variable defined by NerveCenter. The functions and variables available to you are summarized in a pop-up menu for Perl accessible via a right mouse click from the poll condition editing area. (See the following section, [Using the Pop-Up Menu for Perl on page 182](#), for more information.)

For all the details about writing a poll condition, see the following sections:

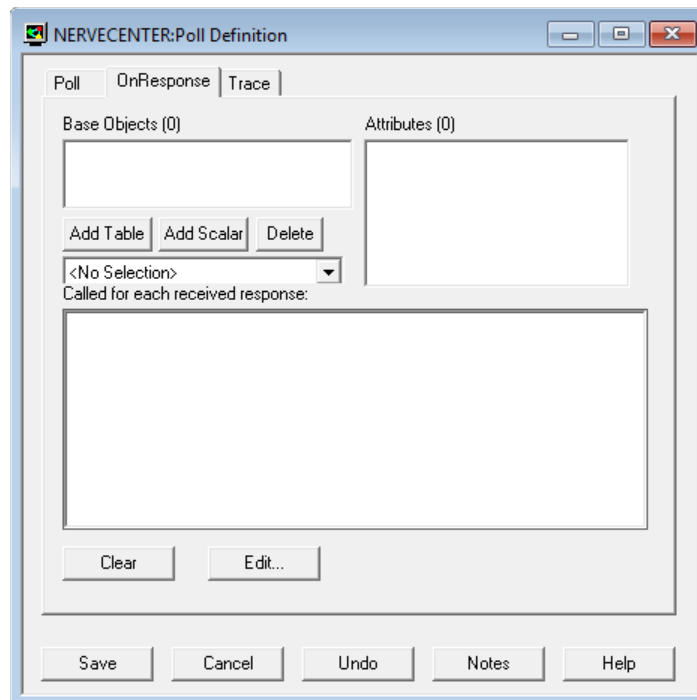
- [The Basic Procedure for Creating a Poll Condition on the facing page](#)
- [Functions for Use in Poll Conditions on page 168](#)
- [NerveCenter Variables on page 301](#)
- [Using the Pop-Up Menu for Perl on page 182](#)
- [Examples of Poll Conditions on page 184](#)

The Basic Procedure for Creating a Poll Condition

The section explains how to use the Poll Condition page in the Poll Definition window to create a poll condition.

TO CREATE A POLL CONDITION

1. In the Poll Definition window, select the **OnResponse** tab.



2. From the **Add Table** or **Add Scalar** buttons, select a base object whose attributes you will use in the poll condition.

You may add any number of Scalar objects to a poll, and any number of Table objects provided they have the same indexing. After adding objects to the list, selecting one will display the object's attributes in the Attributes list.

Note: Selecting the **nl-ping** base object causes NerveCenter to poll using an ICMP Echo (aka ping) instead of SNMP. The attributes of nl-ping, shown in the Attributes list, can be used to examine the outcome of the ICMP operation. See [The nl-ping Property on page 138](#) for more information.

3. If you want to use the shared Perl interpreter, select the **Execute Perl in Global Space** checkbox on the poll definition's Poll property sheet.

Note: If you select Execute Perl in Global Space, the poll condition executes in a shared Perl interpreter. You can use Global variables in your poll condition to share information between other Perl routines such as trigger functions or Perl subroutines, however, Perl intensive poll conditions may impede NerveCenter's performance.

If you do not select Execute Perl in Global Space, the poll condition executes in a Perl interpreter dedicated to poll conditions. This will improve NerveCenter's performance, however you cannot use global variables in your poll condition to share information between other Perl routines such as trigger functions or Perl subroutines.

For more information about the various Perl interpreters, see [NerveCenter and Perl on page 45](#).

4. Place your cursor in the **Poll Condition** text area, and enter the poll condition.

You can enter the poll condition by simply typing the condition in this text area. However, you can also use several shortcuts to enter text:

- One useful shortcut allows you to enter a MIB base object plus an attribute (connected by a period) at the point of the cursor. To use this shortcut, position your cursor where you want to enter the text, and double-click an attribute in the **Attribute** list. (You must have selected a base object from the Base Object drop-down list while the poll condition editing area was empty.)
- You can enter a Perl operator, a call to a NerveCenter function, or a NerveCenter variable using the poll-condition pop-up menu for Perl. To bring up this menu, click the right mouse button while your cursor is in the poll-condition editing area.

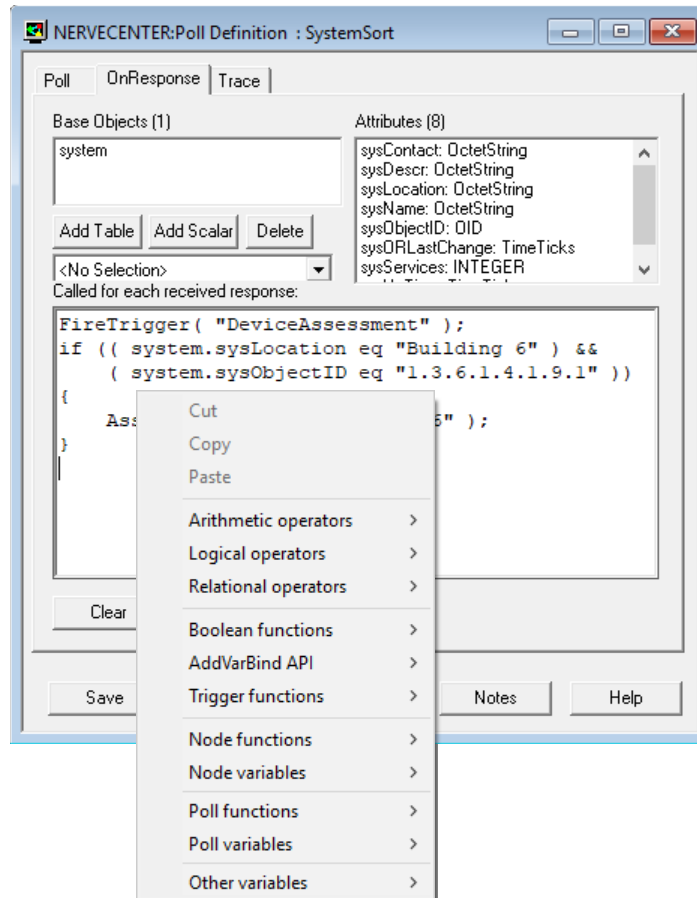
See the section [Using the Pop-Up Menu for Perl on page 182](#) for further information about this pop-up menu.

- You can paste text from the clipboard into the text area.

Functions for Use in Poll Conditions

NerveCenter includes a number of functions that you can use in constructing a poll condition. Several of these functions are designed specifically for use in poll conditions. For example, they enable you to determine the exact number of seconds between polls or to determine the change in the value of a MIB variable between one poll and the next. You can also use the functions `DefineTrigger()`, `FireTrigger()`, `AssignPropertyGroup()`, and `in()` and a set of string-matching functions. These functions can be used not only in defining poll conditions, but in defining other objects as well.

The Perl functions and variables available to you for use in poll conditions are accessible via a right mouse click from the poll condition editing area in the Poll Condition page of the Poll Definition window. (See the section, [Using the Pop-Up Menu for Perl on page 182](#), for more information.)



For detailed information about all of these functions, see the following sections:

- [NerveCenter Functions for Poll Conditions on the next page](#)
- [AddNode\(\) Function on page 174](#)
- [AssignPropertyGroup\(\) Function on page 175](#)
- [DefineTrigger\(\) Function on page 175](#)
- [FireTrigger\(\) Function on page 177](#)
- [in\(\) Function on page 179](#)
- [NC::AlarmCounters on page 179](#)
- [String-Matching Functions on page 181](#)

NerveCenter Functions for Poll Conditions

The functions discussed below are designed specifically for use in poll conditions.

delta()

Usage: OnResponse

Syntax: delta(baseObject.attribute)

Arguments: *baseObject.attribute* - The name of a baseObject as found in the poll's list of Base Objects, and an attribute of that baseObject, as listed in the poll's list of Attributes for the selected baseObject. For example: ifEntry.ifType or system.sysDescr.

Description: Returns the difference between the present value of the object and its prior value, as retrieved by the previous run of this poll. For example, if ifEntry.ifInOctets is currently 10 but previously was 6, then delta(ifEntry.ifInOctets) will return 4.

Example: This statement fires a trigger if the number of SNMP messages sent to a node without an acceptable community name has increased:

```
if (delta(snmp.snmpInBadCommunityNames) >= 1) {
    FireTrigger("authFail");
}
```

elapsed

Usage: OnResponse

Syntax: elapsed

Arguments: None — also omit the typical () characters that follow the function name.

Description: Returns the number of seconds that elapsed between the previous poll and the current poll.

Example: This statement fires a trigger if the poll detects interface traffic levels exceeding 80 percent of capacity:

```
if (((delta(ifEntry.ifInOctets) + delta(ifEntry.ifOutOctets))
* 8) / (ifEntry.ifSpeed * elapsed) >= 0.801) {
    FireTrigger("highLoad");}
```

end_poll()

Usage: OnStart, OnResponse

Syntax: end_poll()

Arguments:None.

Description: For polls that are examining a table, you can skip reading the remainder of the table by calling end_poll(). Processing proceeds directly to the OnEnd expression, if one has been defined. Note this can be called from either the OnStart or OnResponse expression. If called from OnStart, then the agent will not be polled and processing will proceed next directly to the OnEnd expression, if it has been provided.

Example: This statement, as for a OnResponse function, causes the table polling to stop upon the first discovery of an interface that is in the 'down' state.

```
if ( ifEntry.ifOperStatus == down ) {
    FireTrigger( "atLeastOneDownInterface" );
    end_poll();
}
```

jump_to()

Usage: OnResponse

Syntax: jump_to(indexexpression)

Arguments:The value of a possible index into a table baseObject of this poll.

Description: For polls that are examining a table, you can cause the OnResponse to jump forward to the given index. For example: when reading from ifTable, if you are presently at index 3 and you want to skip processing any more rows with OnResponse until at least index 900 is reached, call "jump_to(900);". Note: you can only jump to a higher indexexpression. Meaning, that if you are currently in ifTable at index 100, "jump_to(50);" will be ignored. (Tables in SNMP are always returned in ascending sequence, based on the value range of the table's index expression.)

not_present()

Usage: OnResponse

Syntax: not_present(baseObject.attribute)

Arguments: *baseObject.attribute* - The name of a baseObject as found in the poll's list of Base Objects, and an attribute of that baseObject, as listed in the poll's list of Attributes for the selected baseObject. For example: ifEntry.ifType or system.sysDescr.

Description: Returns 1 (true) or 0 (false), indicating the inverse of present(). A return of 1 (true) indicates the poll was not able to retrieve the given baseObject.attribute expression.

Example: This statement checks whether the attempt to read sysDescr from the agent has failed.

```
if not_present(system.sysDescr) {  
    FireTrigger("noAgent");  
}
```

Note: The previous form of this function stated the baseObject.attribute expression first, followed by the function name. This is still supported, but is not recommended.

```
if (system.sysDescr not_present) {  
    FireTrigger("noAgent");  
}
```


present()

Usage: OnResponse

Syntax: present(baseObject.attribute)

Arguments:*baseObject.attribute* - The name of a baseObject as found in the poll's list of Base Objects, and an attribute of that baseObject, as listed in the poll's list of Attributes for the selected baseObject. For example: ifEntry.ifType or system.sysDescr.

Description: Returns 1 (true) or 0 (false), indicating whether the polled SNMP agent returns a value for this baseObject.attribute expression.

Example: This statement fires a trigger if the poll is able to retrieve the value of ifInUcastPkts from the agent.

```
if ( present( ifEntry.ifInUcastPkts ) ) {  
    FireTrigger( "gotInUcastPkts" );  
}
```

Note: The previous form of this function stated the baseObject.attribute expression first, followed by the function name. This is still supported, but is not recommended.

```
if ( ifEntry.ifInUcastPkts present ) {  
    FireTrigger( "gotInUcastPkts" );  
}
```

previous()

Usage: OnResponse

Syntax: previous(baseObject.attribute)

Arguments:*baseObject.attribute* - The name of a baseObject as found in the poll's list of Base Objects, and an attribute of that baseObject, as listed in the poll's list of Attributes for the selected baseObject. For example: ifEntry.ifType or system.sysDescr.

Description: Returns the prior value of the given baseObject.attribute expression. For example, if ifEntry.ifInOctets is currently 10 and the previous value was 6, then previous(ifEntry.ifInOctets) returns 6.

start_at()

Usage: OnStart

Syntax: start_at(indexexpression)

Arguments:The value of a possible index into a table baseObject of this poll.

Description: For polls that examine a table object, using this function in the poll's OnStart expression allows polling to start at the expressed index (instead of at the beginning of the table). For example, ifTable uses a single integer as its index; if you want the poll to always start at index 100, then in the OnStart property sheet for the poll state “ start_at(100); “. Should there not be at that moment an exact match for the expression, then polling starts at the first available index. For example, if a device's ifTable were to have only index values that were multiples of 8, then “ start_at(20) “ would cause the poll to attempt to start at index 20 but would actually start at index 24.

AddNode() Function

The AddNode() function adds a node to the NerveCenter managed node list. This function can be called from a poll condition, trap mask trigger function, or a Perl Subroutine alarm action.

Syntax: AddNode(“node name”);

Arguments: node name is the IP address of the node to add to the managed node list and must be a valid IP address enclosed in quotes, “123.123.123.123”.

Description: Adds a node to the NerveCenter managed node list, with the following attributes:

- The address assigned to the node will be the address specified for the node name.
- The node property group will be assigned to the “NCDefaultGroup”
- The node community string will be assigned the community string of the node for which the poll condition was executed.
- The node will be marked as 'managed' and 'not suppressed'
- The SNMP version of the node will be assigned the version of the node for which the poll condition was executed.
- No action is taken if a node of the same name already exists in the node list. No validation that the node name doesn't already exist is performed at compile time.

AssignPropertyGroup() Function

You use the `AssignPropertyGroup()` function to assign a property group to a node. The function can be called from a poll condition, a trap mask trigger function, or a Perl Subroutine alarm action. The node affected is the node being polled, the node from which a trap arrived, or the node associated with the trigger that caused an alarm transition (in the case of a Perl Subroutine action).

The syntax of the `AssignPropertyGroup()` function is shown below:

AssignPropertyGroup()

Syntax: `AssignPropertyGroup("propertyGroup")`

Arguments:

propertyGroup - The name of an existing property group.

Description: The function assigns a property group to a node.

Example: The example below shows the `AssignPropertyGroup()` function being used in a Perl Subroutine alarm action. If the variable `$DestStateSev` (which holds the name of the NerveCenter severity of the destination state) contains the string "Critical," the property group of the node associated with the trigger that caused the alarm transition is changed to `CriticalGrp`. The node will now be managed by a new set of behavior models.

```
if ($DestStateSev eq "Critical") {  
    AssignPropertyGroup("CriticalGrp")  
}
```

DefineTrigger() Function

The `DefineTrigger()` function enables you to create triggers which you can assign to variables and fire using `FireTrigger()` in NerveCenter Perl expressions. (In the scope of a subroutine, Perl requires you to define a variable before you can use it.)

You can use `DefineTrigger()` in NerveCenter anywhere that you write Perl expressions (except for Action Router rule conditions):

- Poll conditions
- Perl Subroutine alarm actions
- Mask trigger functions
- OpC mask trigger functions

As with triggers created with `FireTrigger()`, the triggers you create with `DefineTrigger()` are available in the trigger lists NerveCenter displays when you are defining alarm transitions, Perl subroutines, and Action Router rule conditions.

The syntax for the DefineTrigger() function is shown below:

DefineTrigger()

Syntax: DefineTrigger("name")

Arguments:

name - The name of the trigger in quotation marks.

Note: Trigger names can contain the following types of characters: alphanumeric, underscore, and hyphen. No other characters are allowed. The maximum length for trigger names is 255 characters.

Description: DefineTrigger() creates a trigger which you can assign to a variable and fire using FireTrigger().

Example one: The expression creates a trigger named "hello" which is assigned to a Perl variable "\$trig" and is then fired:

```
$Trig = DefineTrigger("hello")
FireTrigger($trig)
```

Example two: The following code excerpt is from a Perl subroutine (TestParentSetNode) associated with the downstream alarm suppression behavior models shipped with NerveCenter. \$TriggerFlag stores the name of the trigger to be fired which depends on the status of the parent node:

```
DefineTrigger('UnReachable');
DefineTrigger('Down');
DefineTrigger('Testing');
...
if( ($ParentStatus eq "Down" || $ParentStatus eq "UnReachable") &&
    $TriggerFlag eq "NotSet" )
{
    $TriggerFlag = "UnReachable";
}
elsif( $ParentStatus eq "Up" )
{
    $TriggerFlag = "Down";
}
elsif( $ParentStatus eq "Testing" && $TriggerFlag ne "Down" )
{
    $TriggerFlag = "Testing";
}
...
FireTrigger( $TriggerFlag );
```

FireTrigger() Function

The FireTrigger() function enables you to fire a trigger from anywhere in NerveCenter that you write Perl expressions:

- Poll conditions
- Perl Subroutine alarm actions
- Mask trigger functions
- OpC mask trigger functions
- Action Router rule conditions

You specify the name of the trigger and optionally its subobject attribute and node attribute.

Caution: In a poll condition FireTrigger function, the subobject and node values are supplied by the poll and can't be overridden. For this reason, you should not attempt to provide the subobject or node parameter when calling the FireTrigger function from a poll condition.

As with triggers created with DefineTrigger(), the triggers you create with FireTrigger() are available in the trigger lists NerveCenter displays when you are defining alarm transitions, Perl subroutines, and Action Router rule conditions.

The syntax for the FireTrigger() function is shown below:

FireTrigger()

Syntax: FireTrigger("name", [subobject], [node], [delay time])

Arguments:

name - The name of the trigger in quotation marks. Name can also be a Perl variable that is assigned a trigger using the DefineTrigger() function. For example:

```
$var=DefineTrigger("myTrigger");  
FireTrigger($var);
```

Note: Trigger names can contain alphanumeric, underscores, and hyphens. No other characters are allowed. The maximum length for trigger names is 255 characters.

subobject - You can pass a subobject to FireTrigger() in one of two ways.

You can use a string literal, for example, "ifEntry.2".

Second, if you called FireTrigger() from a trigger function or a Perl subroutine, you can use the function VbObject(*n*). This function returns the subobject associated with the *n*th variable binding in a trap or trigger.

Note: When firing a trigger from a mask trigger function, you can pass a subobject using the variable \$DefaultSubobject, which contains the subobject associated with the first variable binding in the trap. \$DefaultSubobject works only from a trap mask trigger function.

node - You can pass a node to FireTrigger in one of three ways.

First, you can use the variable \$NodeName, which is the default for this argument. The variable value depends on the context in which it is used, as shown in [Table 14](#).

Table 14: The Value of \$NodeName

If \$NodeName is used in a ...	Its value is ...
Poll condition	The name of the node that was polled.
Trap mask trigger function	The node name associated with the agent address in an SNMP trap.
Perl subroutine	The trigger's node attribute.

Second, include the name of the node in quotation marks, for example, "MyBestRouter" or "192.168.197.110". This string *must* match the name of the node as it's listed in the NerveCenter Node List window.

Finally, if the node name you want to pass to FireTrigger() is in a trap's or a trigger's variable bindings, you can use the function VbValue(*n*) to retrieve that name. This function returns the value of the *n*th variable binding.

delay time - the number of seconds to wait prior to firing the indicated trigger. The delay time can vary between 0 and 2,147,483,647.

To specify a delay time without a subobject or noade name, use default value placeholders as follows:

```
FireTrigger("myTrigger", $DefaultSubobject, $NodeName, delay time)
```

Description: FireTrigger() creates a trigger with the name, subobject, and node values that you supply.

Example: The following call generates a trigger with the name "trigger" and the default subobject and node:

```
FireTrigger("trigger");
```

in() Function

The `in()` function is available for use in poll conditions, trap mask trigger functions, Perl subroutines, and Action Router rule conditions.

in()

Syntax: `in(scalar, scalar, ...)`

Arguments:

scalar - An scalar value in a set of scalar values (often integers representing interface types).

Description: Returns true if the value of the attribute that precedes the function is found in the set of scalars in parentheses.

Example: This statement fires a trigger if a particular interface is part of a broadcast network:

```
if (ifEntry.ifType in (6,7,8,9,11,12,13,15,26,27)) {  
    FireTrigger("broadcast");  
}
```

NC::AlarmCounters

This is available for use in all Perl subroutines, trap masks, action router, and poll conditions.

Name

`NC::AlarmCounters`

Synopsis

```
NC::AlarmCounters->method(key => string, type => local|global,  
value => integer)
```

Description

A Perl based counter object. `NC::AlarmCounters` enables you to do the following in any Perl code in NerveCenter:

- Increment Alarm Counters by a number other than one
- Decrement Alarm Counters by a number other than one
- Create Alarm Counters
- Set Alarm Counters
- Retrieve Alarm Counters

Method

Method	Functionality
new	<p>Creates/initializes the counter.</p> <p>If no initial value is present, then the counter is set to zero.</p> <p>If the counter is global and it already exists, 'new' creates another view into the same counter</p> <p>Returns a reference to the new object</p>
incr	<p>Increments the counter.</p> <p>If no value is present, then increment the counter by one.</p> <p>Returns the new value of the counter.</p>
decr	<p>Decrements the counter.</p> <p>If no value is present, then Decrement the counter by one.</p> <p>Returns the new value of the counter.</p>
get	Retrieve the specified counter.
set	<p>Assigns a value to a counter.</p> <p>If no value is present, then the counter is set to zero.</p> <p>Returns the new value of the counter.</p>
clear	Deletes counter and frees up memory

Arguments

Parameter	Values	Description
key	string	Optional parameter; If not specified, defaults to \$AlarmInstanceID
type	local or global	<ul style="list-style-type: none"> ■ If 'local' is specified, the \$AlarmInstanceID is appended to the key, thus making it 'local' to the Alarm Instance. ■ If 'global' is specified, no modification is made to the key so that it may be easily accessed via other Alarm Instances. <p>The default setting is 'local'.</p>
value	integer	<p>The value to update the counter with</p> <ul style="list-style-type: none"> ■ If the counter already exists (and this is another view) and no value is present, the existing value is retained. ■ If no value is specified and the counter is new, then it is initialized with zero.

Examples

To use the counter functionality in NerveCenter Perl components, include the following:

```
use NC::AlarmCounters;
```

To create a localized counter with the value of 5. This counter is only available in the alarm instance it was created in.

```
my $local_counter = NC::AlarmCounters->new( value => 5 );
```

To create a global counter named errors. Default value is zero. This counter is available through any Perl component.

```
my $global_counter = NC::AlarmCounters->new( key => 'errors', type
=> 'global' );
```

To create another view at the global counter errors. Modifying \$alternate_counter also affects \$global_counter.

```
my $alternate_counter = NC::AlarmCounters->new( key => 'errors',
type => 'global' );
```

To increment the global counter named error. Store new value in \$current_value. Note that no key is necessary because \$global_counter is already defined with a key upon creation.

```
my $current_value = $global_counter->incr();
```

To decrement the local counter by 5. Note that the counter's locality and key are not necessary, as they were determined at the time of creation.

```
$local_counter->decr( value => 5 );
```

To clear out counter when not needed.

```
$local_counter->clear();
$global_counter->clear();
```

String-Matching Functions

NerveCenter provides four string-matching functions (Perl subroutines), which can be used in poll conditions, trap mask trigger functions, OpC trigger functions, Perl subroutines, and Action Router rules. These functions enable you to determine whether a string contains a substring or a word.

For example, you could use the following poll condition to test a system attribute.

```
if (ContainsWord (system.sysDescr, "Description"))
{
    FireTrigger ("TriggerName")
}
```

Note that the substring is enclosed in quotation marks to denote that this is a literal string, whereas the system attribute system.sysDescr does not require quotation marks.

You can precede any of the functions with a ! to negate the value.

Following is a description of each string-matching function:

CaseContainsString()

Syntax: CaseContainsString(*string*, *substring*)

Description: Returns true if *string* contains *substring*. The match is case sensitive.

CaseContainsWord()

Syntax: CaseContainsWord(*string*, *word*)

Description: Returns true if *string* contains *word*, and *word* begins and ends on a word boundary. The match is case sensitive.

ContainsString()

Syntax: ContainsString(*string*, *substring*)

Description: Returns true if *string* contains *substring*. The match is case insensitive.

ContainsWord()

Syntax: ContainsWord(*string*, *word*)

Description: Returns true if *string* contains *word*, and *word* begins and ends on a word boundary. The match is case insensitive.

Using the Pop-Up Menu for Perl

There are various tasks in NerveCenter that require you to write Perl code:

- Creating a poll condition
- Creating a trap mask trigger function
- Creating an OpC mask trigger function
- Creating a Perl subroutine that will be executed by the Perl Subroutine alarm action
- Creating an Action Router rule condition

For each of these tasks, you can use not only Perl 5, but some NerveCenter functions and variables that are appropriate to the task. For instance, if you're writing a trap mask trigger function, you can use NerveCenter functions to retrieve information about the variable bindings in the trap that caused the trigger function to be called. You can also use NerveCenter variables that contain information about the contents of the trap.

What functions and variables are available to you depends on the task you're performing. Therefore, NerveCenter provides a pop-up menu in the editing area for each task that indicates which functions and variables are applicable in that situation. Figure 26 shows the pop-up menu as it appears in the editing area used to create a trap mask trigger function.

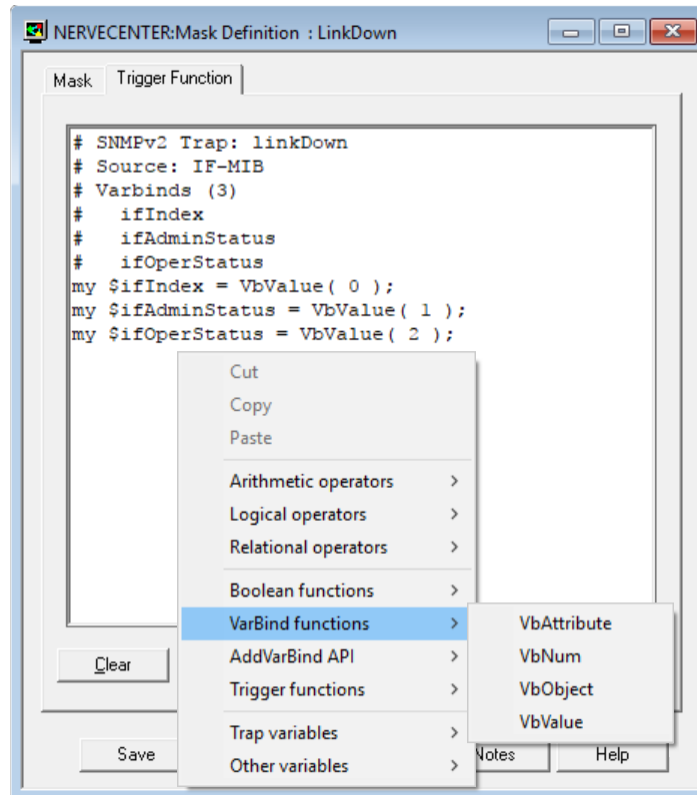


Figure 26: Pop-Up Menu for Perl

The submenu being displayed lists all the variable-binding functions.

Note: In addition to listing NerveCenter functions and variables, the pop-up menus also list Perl's arithmetic, logical, and relational operators.

Besides serving as documentation, these pop-up menus enable you to enter text in an editing area at the point of the cursor. For example, if you were working in the trigger-function window shown above, selecting the menu entry `VbValue` would cause the characters `"VbValue("` to be written to the editing area.

To make this discussion more concrete, let's look at an example. Let's say that you want to write the following trigger function:

```
if ($NodeName ne "troublemaker") {
    FireTrigger("gotIt");
}
```

TO WRITE THIS TRIGGER FUNCTION, YOU WOULD

1. Open the Mask Definition window, and go to the Trigger Function page.
2. Left-click in the Trigger Function editing area, and type **if (**.
3. Press the right mouse button, select the **Trap variables** submenu, and select **\$NodeName** from that submenu.
4. Press the right mouse, select the **Relational operators** submenu, and select **ne** from that submenu.
5. Type **“troublemaker”) {**; then, enter a new line and four spaces.
6. Press the right mouse button, select the **Other functions** submenu, and select **FireTrigger** from that submenu.
7. Type in the remainder of the trigger function.

Examples of Poll Conditions

This section presents a number of sample poll conditions and explains how the poll conditions work.

Example 1

```
if (system.sysLocation eq "Building 6" and
    system.sysObjectID == 1.3.6.1.4.1.9.1) {
    AssignPropertyGroup("Cisco6");
}
```

This poll condition checks to see whether a device is located in Building 6 and whether it is a Cisco product. If the device meets these conditions, it is assigned the property group Cisco6.

Example 2

```
if (ifEntry.ifType present and
    ifEntry.ifSpeed present and
    ifEntry.ifInOctets present and
    ifEntry.ifInUcastPkts present and
    ifEntry.ifInNUcastPkts present and
    ifEntry.ifInDiscards present and
    ifEntry.ifInErrors present and
    ifEntry.ifOutOctets present and
    ifEntry.ifOutUcastPkts present and
    ifEntry.ifOutNUcastPkts present and
    ifEntry.ifOutDiscards present and
    ifEntry.ifOutErrors present) {
    FireTrigger("ifData");
}
```

This poll condition is true as long as the poll is able to read the values of these interface variables from an agent's MIB.

This type of poll condition is useful if you want to gather MIB data that you'll use later in generating a report. For example, if a poll fires an ifData trigger after this poll condition is evaluated, that trigger will contain a list of variable bindings that contains the name and value of each of these attributes. If that trigger causes an alarm transition that has associated with it a Log to File action, these names and values will be written to a log file. That log file can then be used as input to a reporting tool.

Example 3

```
if ((delta(ifEntry.ifInErrors) + delta(ifEntry.ifInDiscards) +
    delta(ifEntry.ifOutErrors) + delta(ifEntry.ifOutDiscards) - 0.05 *
    (delta(ifEntry.ifInErrors) + delta(ifEntry.ifInDiscards) +
    delta(ifEntry.ifOutErrors) + delta(ifEntry.ifOutDiscards) + delta
    (ifEntry.ifInUcastPkts) + delta(ifEntry.ifInNUcastPkts) +
    delta(ifEntry.ifOutUcastPkts) + delta(ifEntry.ifOutNUcastPkts))
    > 0) == 1) {
    FireTrigger("highErrorRate");
}
```

This poll condition is true if the percentage of discarded packets on an interface is greater than five percent during a given polling interval. This is a good example of how to use the delta function.

Example 4

```
if (ifEntry.ifType in (37)) {
    FireTrigger("typeATM");
}
```

This poll condition evaluates to true if an interface's `ifType` attribute equals 37. In other words, the condition is true if the interface is an ATM interface. Obviously, this type of poll condition is useful for classifying interfaces.

Example 5

```
if (((delta(ifEntry.ifInOctets) + delta(ifEntry.ifOutOctets) -
0.00125 * elapsed * ifEntry.ifSpeed > 0) &&
(ifEntry.ifType in (6,7,8,9,11,12,13,15,26,27))) == 1 or
((delta( ifEntry.ifInOctets ) + delta(ifEntry.ifOutOctets) -
0.09375 * elapsed * ifEntry.ifSpeed > 0) &&
!( ifEntry.ifType in (6,7,8,9,11,12,13,15,24,26,27))) == 1) {
    FireTrigger("highLoad");
}
```

This poll condition uses the `delta`, `elapsed`, and `in` functions. It determines whether, during the last poll interval, the traffic on an interface on a broadcast network was greater than 1 percent or whether the traffic on an interface on a point-to-point network was greater than 75 percent.

Documenting a Poll

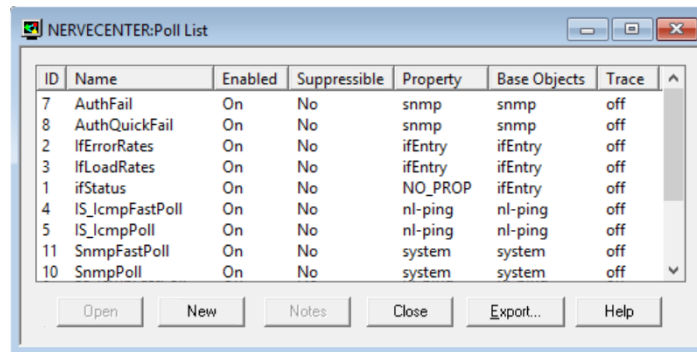
This section explains how to add documentation (notes) to a poll and what should be covered in that documentation.

How to Create Notes for a Poll

You can add notes to a poll by following the procedure outlined in this subsection.

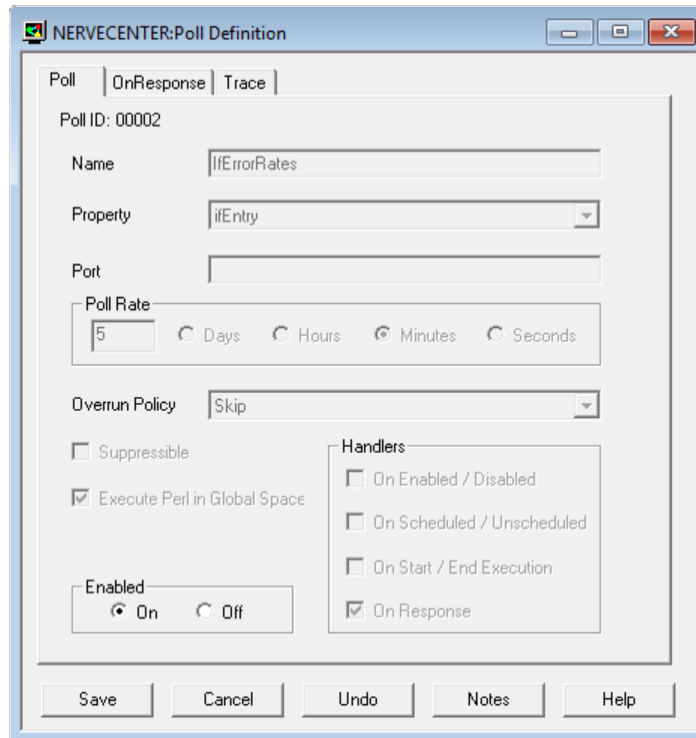
TO ADD NOTES TO A POLL

1. From the client's **Admin** menu, choose **Poll List**.
NerveCenter displays the Poll List window.

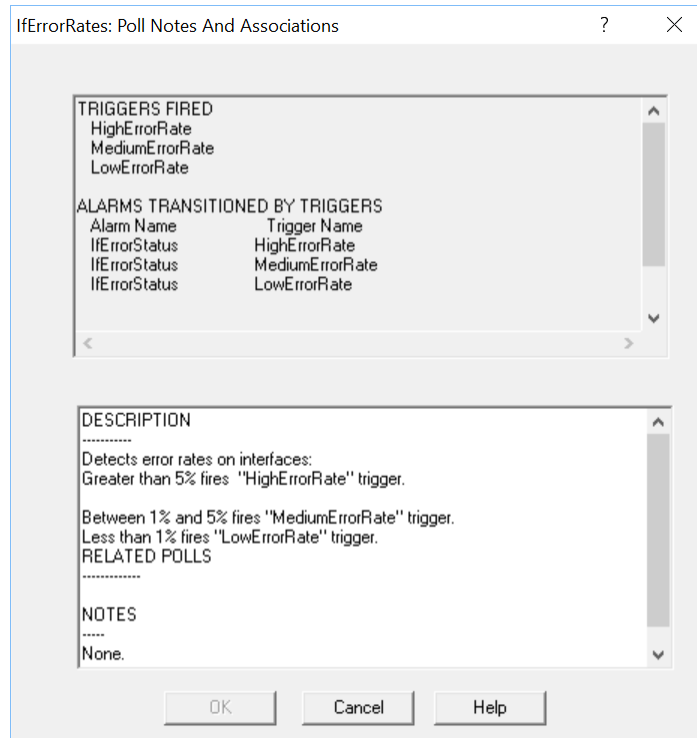


2. Select the poll you want to add a note to from the list.
3. Make sure that your poll is not enabled.

4. Select the **Open** button.
The Poll Definition window is displayed.



- In the Poll Definition window select the **Notes** button.
The Poll Notes and Associations dialog is displayed.



- Enter your documentation for the poll by typing in this dialog. See the section [What to Include in Notes for a Poll on the next page](#) for information on what type of information you should enter here.
- Select the **OK** button at the bottom of the Poll Notes and Associations dialog.
The Poll Notes and Associations dialog is dismissed.
- Select the **Save** button in the Poll Definition window.
Your notes are saved to the NerveCenter database. They can now be read by anyone who opens the definition for your alarm and selects the Notes button.

What to Include in Notes for a Poll

The top pane of the Notes and Associations dialog box contains read-only information about the poll. This data is retrieved from the NerveCenter database and, therefore, may change from time to time when the poll's definition is modified in the database.

This information includes:

- Triggers that are fired by the poll.
- Which alarms are transitioned by these triggers.
- What other NerveCenter objects fire the same triggers.

The bottom pane contains a general description of the poll and any useful information. Users with administrator rights can add or edit this field when creating or customizing polls. Comments should include anything other users might find helpful to know about NerveCenter polls. Following are some suggestions:

- Purpose of the poll
- Associated alarms
- Description of the poll condition
- The poll's property

For example, let's consider the poll definition shown in [Figure 27](#).

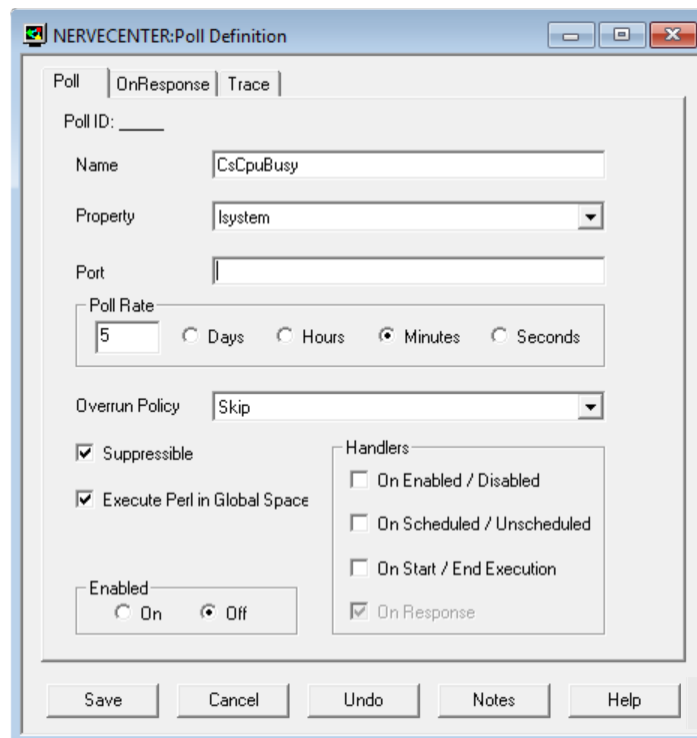


Figure 27: CsCpuBusy Poll

The notes for this poll should look something like this:

Purpose: Detects a busy CPU on a Cisco device

Related alarms: CsCpuUtilization. This alarm tracks CPU utilization on a Cisco device and characterizes it as normal, high, or very high. This poll's trigger, CsCpuBusy, causes a transition from Ground to High.

Poll Condition: If the value of lsystem.avgBusy5 is between 76 and 90, the poll fires its true trigger. The variable avgBusy5 contains an average percentage of CPU utilization. This average is a five-minute exponentially decayed moving average.

Property: lsystem

Enabling a Poll

For a poll to become functional, several conditions must be met:

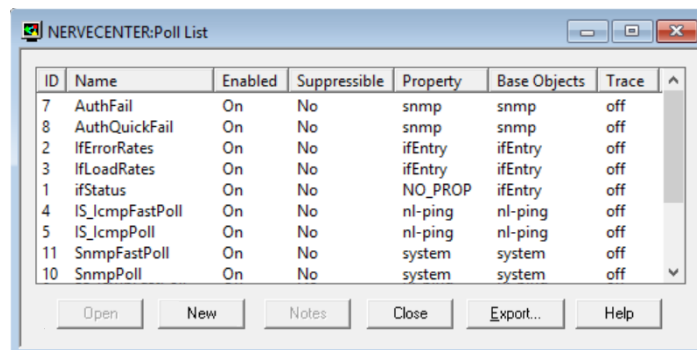
- The poll must be enabled.
- The poll's property must be in the property group associated with one or more nodes, and if those nodes are suppressed, the poll must not be suppressible.
- There must be an enabled alarm with a *pending* state transition that can be affected by the poll.

This section explains how to enable a poll.

TO ENABLE A POLL

1. From the client's **Admin** menu, choose **Poll List**.

NerveCenter displays the Poll List window.



2. Select the poll you want to enable from the list.

The Open button becomes enabled.

3. Select the **Open** button.

The Poll Definition window is displayed and shows the definition of the poll you selected.

NERVECENTER:Poll Definition

Poll | OnResponse | Trace

Poll ID: 00007

Name: AuthFail

Property: snmp

Port:

Poll Rate: 30 Days Hours Minutes Seconds

Overrun Policy: Skip

Suppressible

Execute Perl in Global Space

Enabled: On Off

Handlers:

On Enabled / Disabled

On Scheduled / Unscheduled

On Start / End Execution

On Response

Save Cancel Undo Notes Help

4. Select the **On** radio button in the **Enabled** frame.
5. Select the **Save** button.

The poll is now enabled.

Note: You can also enable a poll by selecting the poll in the Poll List window, pressing the right mouse button while your cursor is over the entry for the poll, and choosing **On** from the popup menu.

Trap masks give you the ability to screen SNMP traps sent by managed nodes and received by NerveCenter for traps of interest. This chapter explains in detail how to define and use trap masks.

About Trap Masks

Figure 28 depicts the role that a trap mask plays in a behavior model.

[review - image?](#)



Figure 28: Role of a Trap Mask in a Behavior Model

Note that a trap mask is like a poll in that it is tied to one or more alarms by the triggers it can fire. If there are no pending alarm transitions that the mask can affect, the mask is disabled in the sense that it will not be applied to any incoming SNMP traps.

Assuming that the mask can affect an alarm transition, the mask is applied to SNMP traps as they arrive and determines whether it should fire a trigger in response to the trap. A mask can fire a trigger in one of two ways:

- A trap mask can fire a simple trigger. A mask designed to fire this type of trigger looks only at the Enterprise, Generic trap, and Specific trap fields in a trap's Protocol Data Unit (PDU). If these fields meet predefined conditions, the mask fires a trigger. All the triggers that this mask ever fires will have the same name.
- A mask can also fire a trigger from a trigger function by calling the FireTrigger() function. This type of mask looks at the fields mentioned above to determine whether it should call its trigger function. If called, this trigger function generally looks at the trap's variable bindings and may fire one of several triggers depending on the contents of the variable bindings.

If a mask fires a trigger, that trigger interacts with the alarm system just as a trigger fired by a poll does. If the necessary attributes of the trigger match the corresponding attributes of a pending alarm transition, a state transition occurs.

How NerveCenter Decodes SNMPv2c/v3 Traps

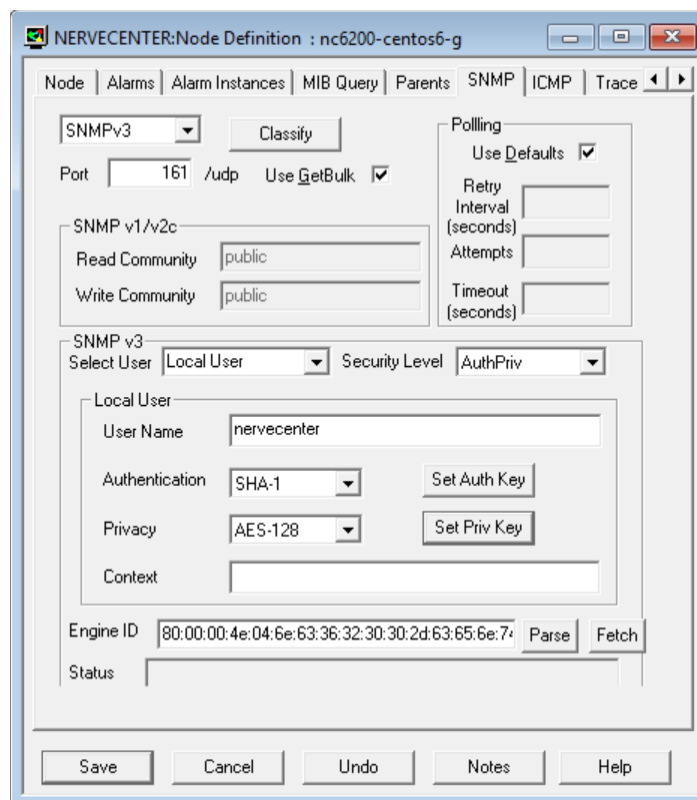
Because SNMPv2c/v3 traps use a different architecture that extends security and administration, the mechanics of how NerveCenter receives an SNMPv2c/v3 trap is different than how it receives an SNMPv1 trap.

When an SNMPv3 trap is received by the NerveCenter Server, it attempts to decode the trap. If the SNMP engine sending the trap is not registered, then NerveCenter installs the engine.

If the user name that is listed in the received trap's header does not match that for the matching node's User Name field, then the trap is not processed and a 'Configuration Mismatch' error is written to the V3 Operations Log.

Next, if the user name matches and the security level of the received trap is other than NoAuthNoPriv, NerveCenter tries to match the authentication value against the selection and key given for the matching node. If the authentication cannot be matched, then the trap is not processed and a 'Configuration Mismatch' error is written to the V3 Operations Log.

Finally, if the authorization portion of the received trap is successful, then NerveCenter checks the trap's v3 Context value. If this does not match the Context setting in the node, then NerveCenter does not process the trap and writes a 'Configuration Mismatch' to the V3 Error Log.



For more information about SNMPv3 in NerveCenter, see "[Configuring SNMP Settings for Nodes](#)" on page 99.

How NerveCenter Decodes ICMP Events

When ICMP event processing is enabled, each ICMP message received is transposed to an SNMPv1 trap and made available for processing. Such traps are built with a defined set of attributes, and the varbinds are prepared using the attributes of the nl-ping base object. This trap, nl-icmp-event, is defined in the NETLABS-PING-MIB file (nl-ping.asn1).

```
nl-icmp-event TRAP-TYPE
    ENTERPRISE openservice          -- "1.3.6.1.4.1.78
    VARIABLES { nl-ping-response, nl-ping-uctype, nl-ping-uccode,
                nl-ping-roundtrip, nl-ping-senderip, nl-ping-origini
    DESCRIPTION "ICMP event parameters formatted as SNMPv1 Trap"
    ::= 1000
```

The definition of nl-icmp-event can be used to construct a trap mask, as per any other defined SNMP notification. Handling and processing proceeds identically to the manner given for SNMP Trap events.

Note: A recommended trap mask definition for handling nl-icmp-event trap occurrences can be found in the importable model ICMP-Event-Processing. This model loads the NC-ICMP-Event trap mask as well as a sample alarm, NC-PingMonitor.

Listing Trap Masks

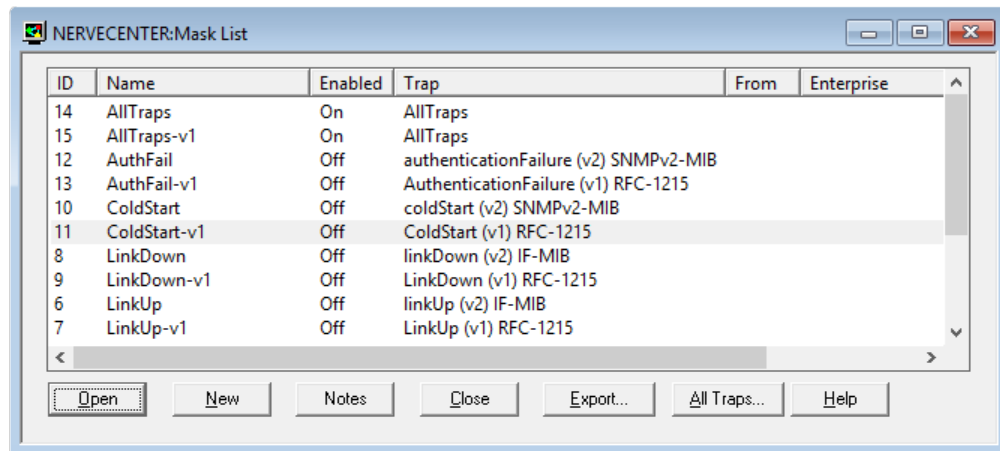
This section explains how to display a list of the trap masks currently defined in the NerveCenter database. The section also explains how to view the definition of a particular trap mask.

For information on creating a new trap mask, see [Defining a Trap Mask on page 198](#).

TO DISPLAY A LIST OF TRAP MASKS AND THEN DISPLAY A PARTICULAR MASK'S DEFINITION

1. From the client's **Admin** menu, choose **Mask List**.

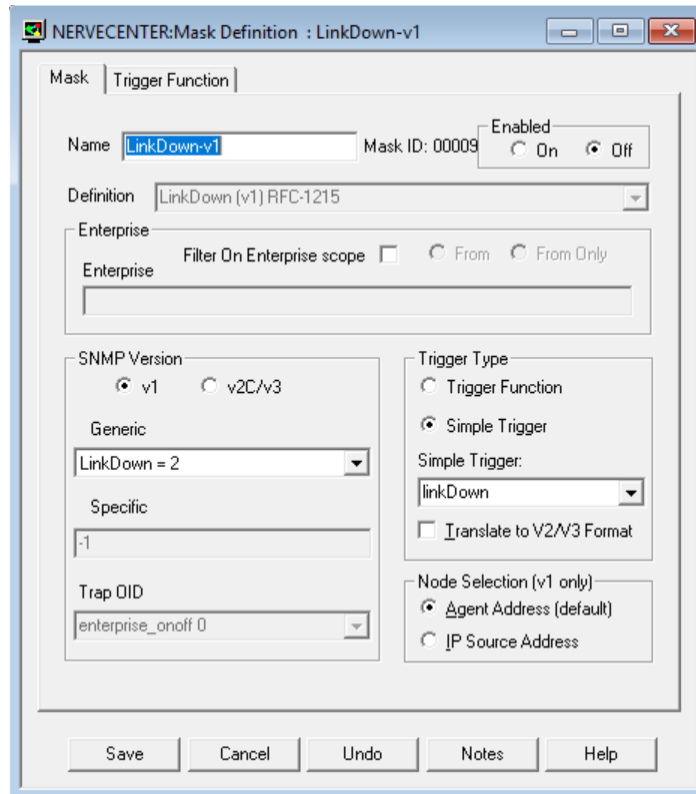
The Mask List window is displayed.



This window lists all NerveCenter masks and provides a brief definition of each. For each mask, the window specifies a name and the following information:

- Whether the mask is currently enabled
 - The generic trap the mask is looking for
 - The enterprise from which the trap must come before the mask will fire a trigger
 - The name of the mask's simple trigger or an indication that the mask uses a trigger function
2. Select a mask from the mask list.

3. Select the **Open** button.
NerveCenter displays the Mask Definition window.



The mask defined in this figure is named LinkDown. It is looking for a generic trap 2 from any managed node and will fire the simple trigger linkDown if it finds one.

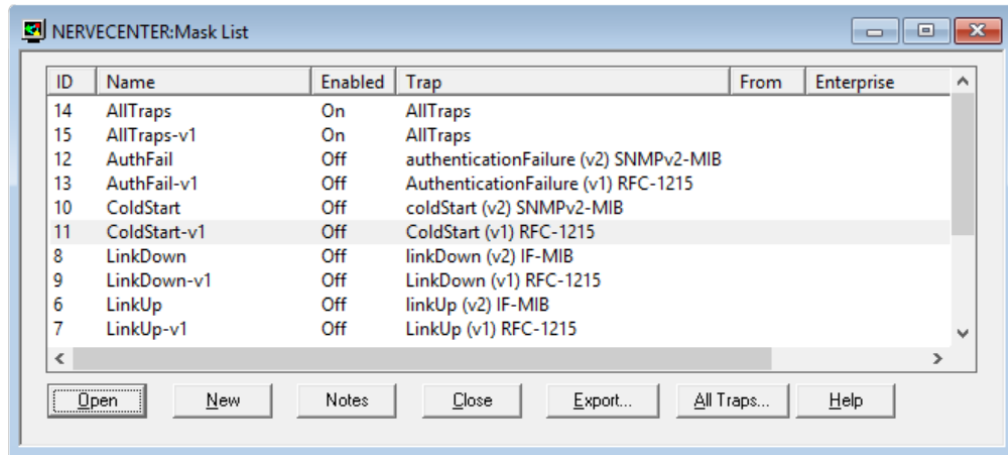
Defining a Trap Mask

This section outlines the procedure for creating a trap mask.

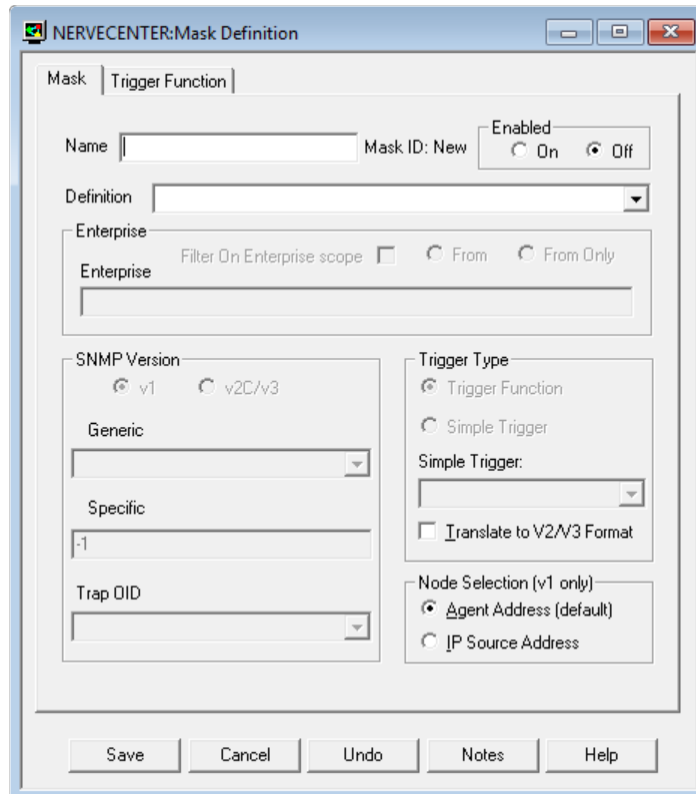
TO DEFINE A NEW TRAP MASK

1. From the client's **Admin** menu, choose **Mask List**.

The Mask List window is displayed.



2. Select the **New** button.
The Mask Definition window appears.



3. In the **Name** text field, type a unique name for the trap mask.

Note: The maximum length for trap mask names is 255 characters. A trap mask name should describe the type of trap the mask is looking for, for example, “ColdStart.”

4. From the **Generic** list box, select a generic trap type.
Before a trap mask can fire a trigger, the value of this field must match the value of a trap’s Generic trap field, which may contain any of the enumeration constants shown in the following table:

Constant	Meaning
coldStart (0)	Signifies that the sending protocol entity is re-initializing itself such that the agent’s configuration or the protocol entity implementation must be altered.
warmStart (1)	Signifies that the sending protocol entity is re-initializing itself such that neither the agent configuration nor the protocol entity implementation is altered.

Constant	Meaning
linkDown (2)	Signifies that the sending protocol entity recognizes a failure in one of the communication links represented in the agent's configuration. The trap PDU of type linkDown contains as the first element of its variable bindings the name and value of the ifIndex instance for the affected interface.
linkUp (3)	Signifies that the sending protocol entity recognizes that one of the communication links represented in the agent's configuration has come up. The trap PDU of type linkUp contains as the first element of its variable bindings the name and value of the ifIndex instance for the affected interface.
authenticationFailure (4)	Signifies that the sending protocol entity is the addressee of a protocol message that is not properly authenticated.
egpNeighborLoss (5)	Signifies that an EGP neighbor for whom the sending protocol entity was an EGP peer has been marked down and that the peer relationship no longer exists. The trap PDU of type egpNeighborLoss contains as the first element of its variable bindings the name and value of the egpNeighAddr instance for the affected neighbor.
enterpriseSpecific (6)	Signifies that the sending protocol entity recognizes that an enterprise-specific event has occurred. The Specific trap field identifies the particular trap that occurred.

Note: The above definitions are taken from RFC1157.

If you select EntSpecific = 6 (an enterprise specific trap), the Specific text field is enabled, and you must enter a vendor-specific trap number in that field.

If you select AllTraps = -1, the mask will disregard the contents of each trap's Generic trap field when looking for traps of interest. That is, any generic trap type in the trap meets the trap mask's requirement.

5. If you want the trap mask to examine the contents of a trap's Enterprise field, follow these directions:
 - a. Select the **Filter on Enterprise scope** checkbox.
Controls in the **Enterprise** group box become enabled.
 - b. Select one of the following radio buttons:
 - **From**—specify that the trap's Enterprise field must contain an OID that either matches the OID in your mask's Enterprise field, or is subordinate to it.
 - **From Only**—indicate that the trap's enterprise must match the mask's enterprise exactly.
 - c. In the **Enterprise** text field, enter an OID, or a name that maps to an OID.
6. If the trap NerveCenter will process is an SNMP version 2c or 3 trap, select the **v2C/v3** radio button.
7. For SNMPv1 traps, if your mask's generic trap type is 6 (enterprise specific), enter a vendor-specific trap number in the **Specific** text field.

Before the mask can fire a trigger, the number you enter in the **Specific** field must match the value of a trap's Specific trap field.

Note: To determine what enterprise specific traps an SNMP agent can produce, consult the vendor's ASN.1 files or other documentation.

8. For SNMPv2c or v3 traps, enter the trap OID.

You can select one of the OID values, choose All Traps, or type the value for a particular enterprise trap OID. SNMPv3 trap OID values map to generic traps as shown below.

Trap	Generic Value	SnmpTrapOID.0
coldStart	0	1.3.6.1.6.3.1.1.5.1
warmStart	1	1.3.6.1.6.3.1.1.5.2
LinkDown	2	1.3.6.1.6.3.1.1.5.3
linkUp	3	1.3.6.1.6.3.1.1.5.4
AuthFail	4	1.3.6.1.6.3.1.1.5.5
EgpNeighLoss	5	1.3.6.1.6.3.1.1.5.6

9. Select one of the **Trigger Type** radio buttons:

- **Simple Trigger**—if the values in your mask's Generic, Enterprise, and Specific fields are sufficient to define the trap you are looking for.
- **Trigger Function**—if you need to specify additional information: for example, the values of variable bindings.

If you select the **Simple Trigger** radio button, the **Simple Trigger** combo box is enabled.

10. In [Step 9](#), if you selected:

- **Simple Trigger**—enter a trigger name in the **Simple Trigger** field. You can either type in the name of a new trigger or choose a trigger from the list of existing triggers.
- **Trigger Function**—perform the following:
 - a. Select the **Trigger** function tab, and enter a trigger function on the **Trigger Function** page. This trigger function is a Perl subroutine that you can use to check the values of variable bindings or examine other pertinent information and to fire appropriate triggers. For complete information on writing trigger functions, see the section [Writing a Trigger Function on the next page](#).
 - b. If you want to use the shared Perl interpreter, select the **Execute Perl in Global Space** checkbox.

Note: If you select Execute Perl in Global Space, the poll condition executes in a shared Perl interpreter. You can use Global variables in your poll condition to share information between other Perl routines such as trigger functions or Perl subroutines, however, Perl intensive poll conditions may impede NerveCenter's performance.

If you do not select Execute Perl in Global Space, the poll condition executes in a Perl interpreter dedicated to poll conditions. This will improve NerveCenter's performance, however you cannot use global variables in your poll condition to share information between other Perl routines such as trigger functions or Perl subroutines.

For more information about the various Perl interpreters, see [NerveCenter and Perl on page 45](#).

11. Select the **Save** button at the bottom of the Mask Definition window to save your mask.

Note: Remember that you must enable the trap mask (by setting Enabled to On) before using it in a behavior model. While the mask is disabled, it is not used in the examination of any incoming traps. This means that any behavior models that use this trap mask as the sole source of triggers are also disabled.

Writing a Trigger Function

If a mask cannot describe the type of trap it is looking for by specifying the Generic trap contents, Enterprise, and Specific trap fields, it must use a trigger function. This Perl function can include additional conditions that the trap must meet, and it can fire different triggers as appropriate.

Most trigger functions are very similar in structure. They follow this pattern:

```
if (condition1) {
    FireTrigger(arguments);
}
elsif (condition2) {
    FireTrigger(arguments);
}
else {
    FireTrigger(arguments);
}
```

The conditions, which can be arbitrarily complex, generally test the contents of a trap's variable bindings. However, they can test other information as well; for example, a condition can determine whether a trap came from a particular node. The FireTrigger() function fires a trigger, whose name, subject, and node you can control.

Note: The maximum length for trigger names is 255 characters.

To assist you in writing trigger functions, NerveCenter provides:

- A set of functions that enable you to examine the contents of a trap's variable bindings and to fire triggers, among other things
- A set of predefined variables that give you access to information about the trap you're examining, such as the community string in the trap's SNMP message
- A pop-up help menu in the trigger function editing area that lists all the NerveCenter functions and variables available for use in a trigger function.

For further information about these predefined functions and variables and the pop-up help menu, see the following sections:

- [Functions for Use in Trigger Functions below](#)
- [Variables for Use in Trigger Functions on page 207](#)
- [Using the Pop-Up Menu for Perl on page 182](#)

Also, see the section [Examples of Trigger Functions on page 208](#). This section presents several sample trigger functions that show a number of the functions and variables being used in context.

Functions for Use in Trigger Functions

NerveCenter provides a number of functions (actually Perl subroutines) that facilitate the writing of trigger functions. The list below indicates what types of functions are available and where you can find detailed information about each function:

- **Variable-binding functions.** These functions enable you to determine the number of variable bindings in a trap's variable-binding list and to obtain information about each variable binding. For instance, you can retrieve the subobject and attribute associated with a variable-binding object and the value of a variable-binding object.

For reference information about these functions, see the section [Variable-Binding Functions on the next page](#).

- **AddNode().** This function enables you to add a node to the NerveCenter managed node list from a NerveCenter Perl expression.

For reference information about this function, see the section [AddNode\(\) Function on page 206](#).

- **String-matching functions.** These functions enable you to determine whether a string contains another string or a particular word. The functions are useful in conditions that test the value of a variable binding for a substring.
 For reference information about these functions, see the section [String-Matching Functions on page 181](#).
- **DefineTrigger().** This function enables you to create triggers which you can assign to variables and fire using FireTrigger() in NerveCenter Perl expressions.
 For reference information about this function, see the section [DefineTrigger\(\) Function on page 175](#).
- **FireTrigger().** This function enables you to fire a trigger from your trigger function. You can specify the name, subobject, and node attributes of the trigger.
 For reference information about this function, see the section [FireTrigger\(\) Function on page 177](#).
- **AssignPropertyGroup().** This function enables you to assign the node that sent a trap to a property group.
 For reference information about this function, see the section [AssignPropertyGroup\(\) Function on page 175](#).
- **NC::AlarmCounters.** This function enables you to do any of the following: increment alarm counters by a number other than one, decrement alarm counters by a number other than one, create alarm counters, set alarm counters, and retrieve alarm counters. For reference information about this function, see the section [NC::AlarmCounters on page 179](#).
 NC::AlarmCounters are independent of and not related to the alarm action Alarm Counter. See [Alarm Counter on page 271](#) for details.
- **in().** This function enables you to determine whether one scalar value is in a set of scalar values.
 For reference information about this function, see the section [in\(\) Function on page 179](#).

Variable-Binding Functions

Before looking at the variable-binding functions, let's make sure that we're using the same terminology.

When a trap arrives, NerveCenter looks at the trap's variable bindings and, for each variable binding, it sees an object and a value.

Object	Value
1.3.6.1.2.1.1.1.0	"Windows Workstation"

Figure 29: Variable Binding

In this case, the object is the OID encoding of the object type (sysDescr) plus an instance, and the value is a string that describes the system.

When NerveCenter sees this variable binding, it stores the following information. The portion of the OID that corresponds to the system group is stored as the binding's *base object*, and the instance (0) is stored as the binding's *instance*. When concatenated, the base object and the instance form what NerveCenter calls a *subobject*.

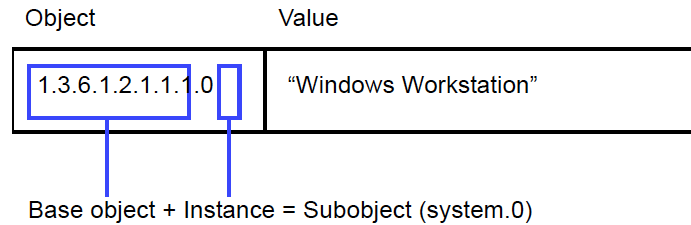


Figure 30: Base Objects, Instances, and Subobjects

The variable `sysDescr` is stored as the binding's *attribute*.

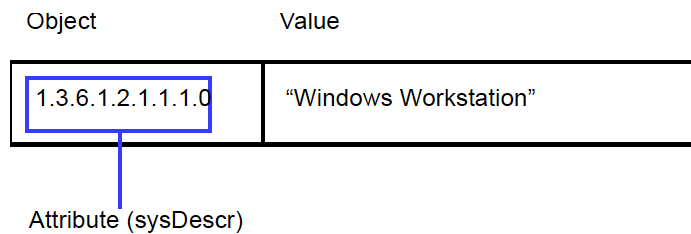


Figure 31: Attributes

Finally, the value "Windows Workstation" is stored as the binding's *value*.

The variable-binding functions give you access to a binding's subobject, attribute, and value. There's also a function that returns the number of variable bindings in a trap or trigger.

Each of the variable-binding functions is explained below:

VbAttribute()

Syntax: `VbAttribute(index)`

Description: Returns the attribute from the variable binding with an index of *index*. The first variable binding has an index of 0.

VbNum()

Syntax: `VbNum()`

Description: Returns the number of variable bindings in the trap's variable-binding list.

VbObject()

Syntax: VbObject(*index*)

Description: Returns the subobject from the variable binding with an index of *index*. The first variable binding has an index of 0.

VbValue()

Syntax: VbValue(*index*)

Description: Returns the value from the variable binding with an index of *index*. The first variable binding has an index of 0.

AddNode() Function

The AddNode() function adds a node to the NerveCenter managed node list. This function can be called from a poll condition, trap mask trigger function, or a Perl Subroutine alarm action.

Syntax: AddNode("*node name*");

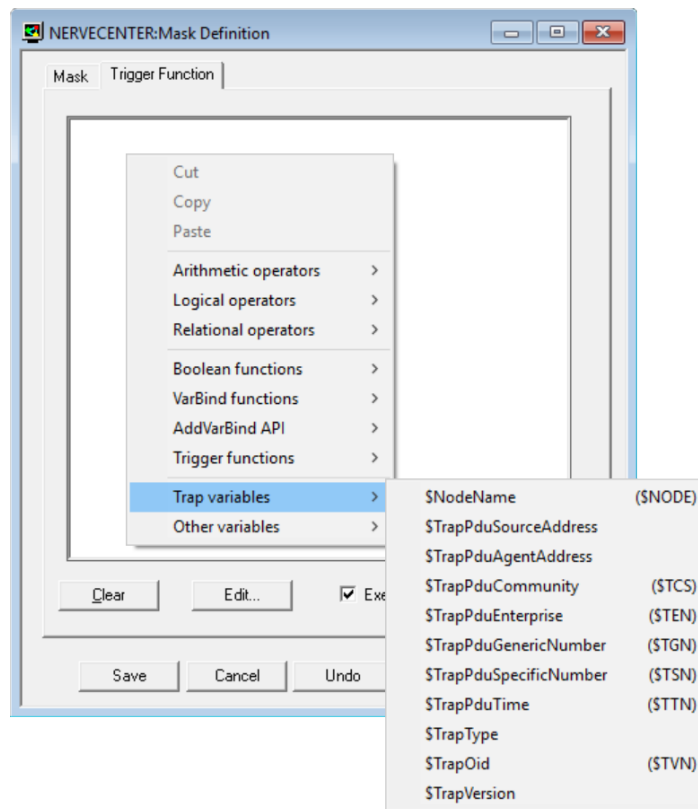
Arguments: node name is the IP address of the node to add to the managed node list and must be a valid IP address enclosed in quotes, for example, "123.123.123.123".

Description: Adds a node to the NerveCenter managed node list, with the following attributes:

- Node name must be a valid IP address enclosed in quotes (for example "192.168.1.1").
- The address assigned to the node will be the address specified for the node name.
- The node property group will be assigned based on the enterprise ID indicated by the trap.
- The node community string will be assigned based on the community indicated by the trap.
- The node will be marked as 'managed' and 'not suppressed'
- The SNMP version of the node will be assigned based on the version of the trap received. A version 1 trap will create a version 1 node, a V2C trap will create a V2C node.
- No action is taken if a node of the same name already exists in the node list. No validation that the node name doesn't already exist is performed at compile time.

Variables for Use in Trigger Functions

NerveCenter defines several variables for use in trap mask trigger functions. For the most part, these variables contains the values of the fields in a trap's Protocol Data Unit (PDU), with the exception of the variable bindings.



The complete list of variables that you can use in a trap mask trigger function is shown in [Table 15](#)

Table 15: Variables Used in Trigger Functions

Variable	Description
\$NodeName	The name of the node that was the source of the trap
\$TrapOid	The TrapOID value of a SNMPv2c/v3 Trap or Inform.
\$TrapPduAgentAddress	The IP address of the SNMP agent that sent the trap
\$TrapPduCommunity	The community name included in the SNMP message
\$TrapPduEnterprise	An OID representing the object that generated the trap
\$TrapPduGenericNumber	The generic trap type
\$TrapPduSourceAddress	The IP Address given in the received trap messages IP Header.
\$TrapPduSpecificNumber	A specific trap code
\$TrapPduTime	The time, in hundredths of a second, between the last initialization of the network entity and the generation of the trap
\$TrapType	Will be either "Trap" or "Inform", depending on the received trap's type.
\$TrapVersion	Will be one of "1", "2" or "3" depending on the version of SNMP found in received trap.

Examples of Trigger Functions

This section presents several trigger functions and explains what the functions do.

Example 1

```
if ($NodeName ne "troublemaker") {
    FireTrigger("gotIt");
}
```

If the node that sent the trap is any node except troublemaker, issue a trigger named gotIt. This example would be useful if you had a device sending inappropriate traps. The trigger function would allow you to pay attention to a trap only when it came from other, more dependable, devices.

Example 2

```
if (system.sysContact eq "Tom Jones") {
    FireTrigger("jonesJob");
} else {
    FireTrigger("otherAdmin");
}
```

If the first variable binding containing the sysContact attribute has the value "Tom Jones," a jonesJob trigger is issued. Otherwise, an otherAdmin trigger is issued.

Example 3

```
if (snmp.snmpInBadCommunityNames > 25) {
    FireTrigger("tooManyIntrusions", VbObject(2));
}
```

If the `snmpInBadCommunityNames` attribute is found in one of the variable bindings, its value is checked. If there were at least 26 attempts to communicate with the trap's node without the proper community string before the trap was issued, a `tooManyIntrusions` trigger is issued. The subobject assigned to the trigger is the subobject associated with the third variable binding.

This would be an effective way to ignore authorization traps until they became significant.

Example 4

```
if (ContainsString(VbValue(2)), "crucial message") {
    FireTrigger("trig");
}
```

If the third variable binding, assumed here to be defined as a `DisplayString`, contains the string "crucial message," the trigger `trig` is generated. This type of trigger function is useful when text messages are sent to NerveCenter via traps.

Example 5

```
if ((VbNum() == 5) && (.8 * VbValue(3) < VbValue(4))) {
    FireTrigger("diskSpaceLow", VbObject(1));
} elseif ((VbNum() == 4) && (VbValue(3) > 400000000)) {
    FireTrigger("diskSpaceLow", VbObject(1));
}
```

This example assumes that there is an enterprise-specific trap that contains information about disk space use. An older version of the vendor's agent sent a trap with four variable bindings, the last variable binding containing the amount of disk space used (`VbValue(3) > 400000000`). A newer version of the agent sends traps with five variable bindings: the last binding contains disk space used, and the next to last contains the disk space capacity. If a trap arrives from a newer agent, you want to fire a trigger only if available disk space is less than 20 percent. This trigger function not only enables you to ignore noncritical situations, but handles all releases of your vendor's device.

Example 6

```
if (VbValue(0) == 1) {
    FireTrigger("thisProblem", VbObject(2), VbValue(1));
} elseif (VbValue(0) == 2) {
    FireTrigger("thatProblem", VbObject(2), VbValue(1));
} elseif (VbValue(0) == 3) {
    FireTrigger("otherProblem", VbObject(2), VbValue(1));
} else {
    FireTrigger("huhProblem", VbObject(2), VbValue(1));
}
```

This example illustrates how to deal with a class of traps sent by some vendors in which the trap's source and specific number are constant. These vendor's agents insert a problem identifier and the source of the problem into the trap's variable bindings. This example assumes that the problem identifier is in the first variable binding, the source node is in the second, and any other associated data follows in successive positions.

Documenting a Trap Mask

This section explains how to add documentation (notes) to a trap mask and what should be covered in that documentation.

- [How to Create Notes for a Trap Mask on the facing page](#)
- [What to Include in Notes for a Trap Mask on page 214](#)

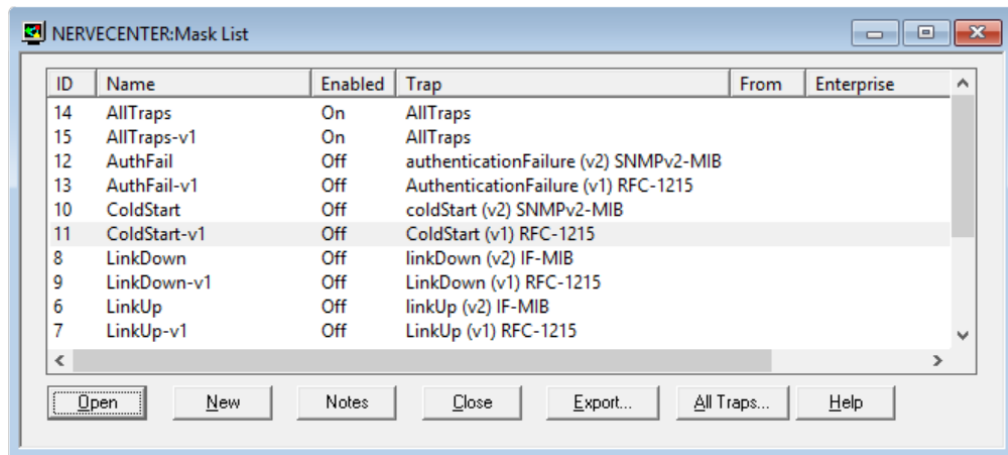
How to Create Notes for a Trap Mask

You can add notes to a trap mask by following the procedure outlined in this subsection.

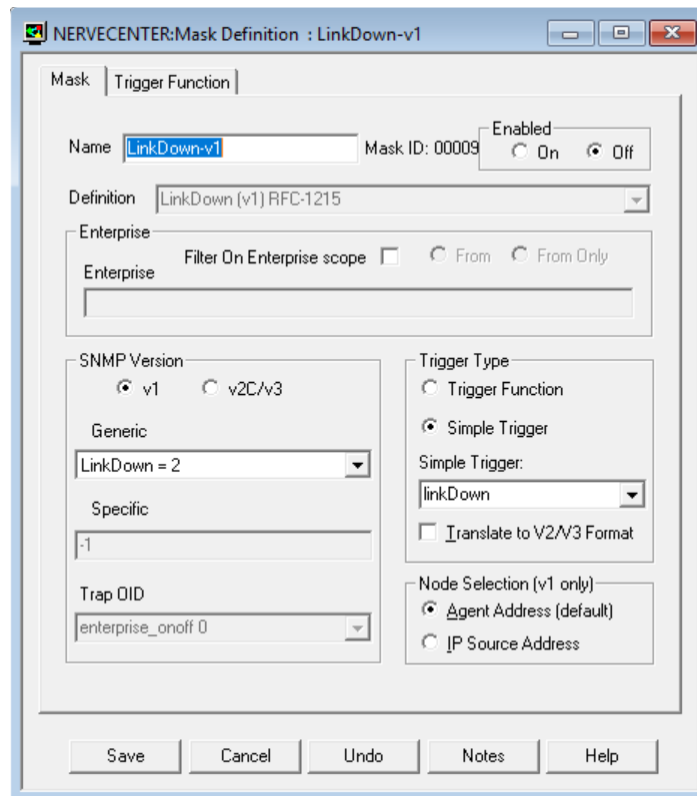
TO ADD NOTES TO A TRAP MASK

1. From the client's **Admin** menu, choose **Mask List**.

NerveCenter displays the Mask List window.

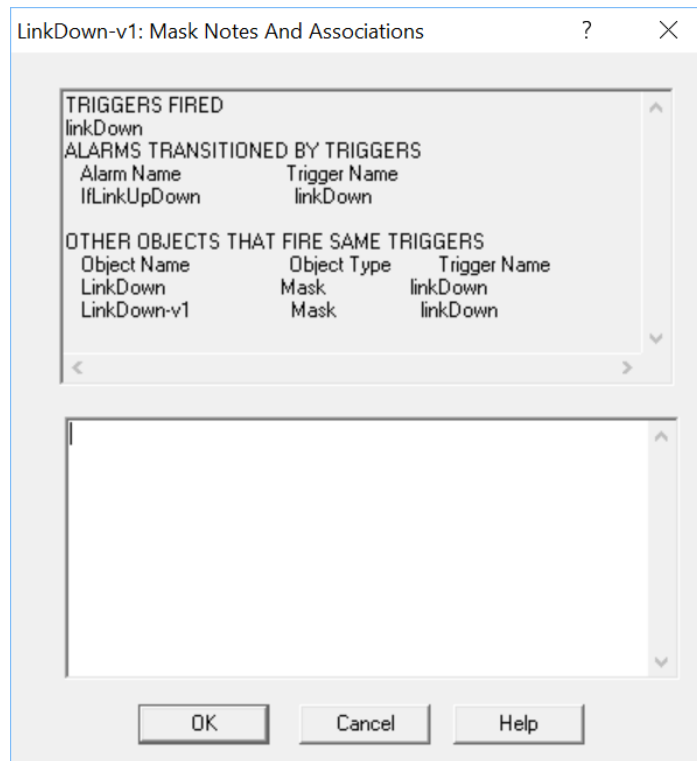


2. Select the **Open** button.
The Mask Definition window appears.



3. Make sure that your mask is not enabled.

- In the Mask Definition window, select the **Notes** button.
The Mask Notes window is displayed.



- Enter your documentation for the trap mask by typing in this window. See the section [What to Include in Notes for a Trap Mask on the next page](#) for information on what type of information you should enter here.
- Select the **OK** button at the bottom of the Mask Notes window.
The Mask Notes window is dismissed.
- Select the **Save** button in the Mask Definition window.
Your notes are saved to the NerveCenter database. They can now be read by anyone who opens the definition for your mask and selects the Notes button.

What to Include in Notes for a Trap Mask

The top pane of the Notes and Associations dialog box contains read-only information about the mask. This data is retrieved from the NerveCenter database and, therefore, will change when the mask's definition is modified in the database.

This information includes:

- Triggers that are fired by the mask.
- Which alarms are transitioned by these triggers.
- What other NerveCenter objects fire the same triggers.

The bottom pane contains a general description of the mask and any useful information. Users with administrator rights can add or edit this field when creating or customizing masks. Comments should include anything other users might find helpful to know about NerveCenter masks. Following are some suggestions:

- Provide a brief description of the mask, including its purpose and function.
- List vendor-specific information, if applicable.
- State the conditions under which the trigger will fire an alarm instance. If a trigger function is defined, provide a brief description of the function.

We recommend that you include the following information in the notes for your trap mask:

- Purpose of the mask
- Associated alarms
- Vendor-specific information (if appropriate)
- Description of the trigger function (if appropriate)

For example, let's consider the trap mask shown in Figure 32 and Figure 33

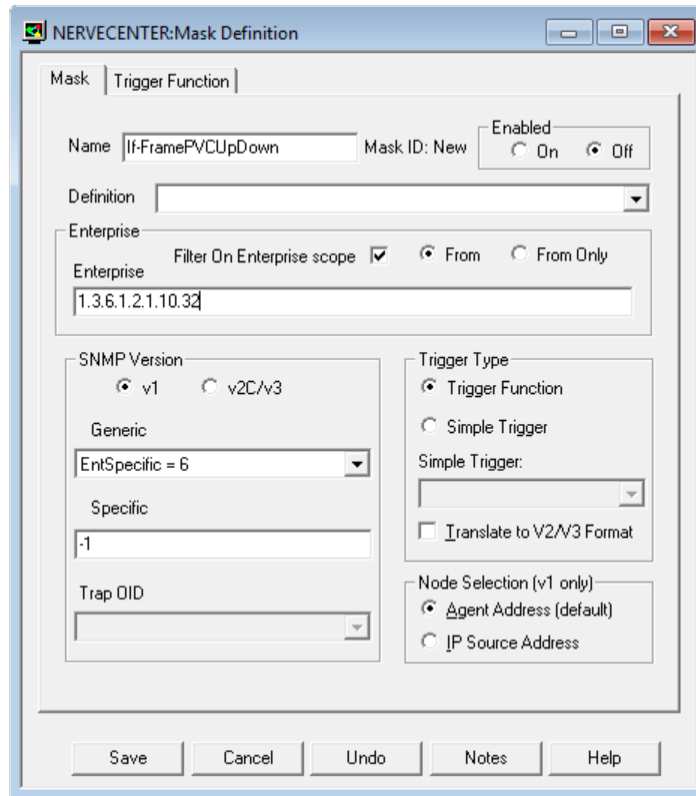


Figure 32: Basic Definition

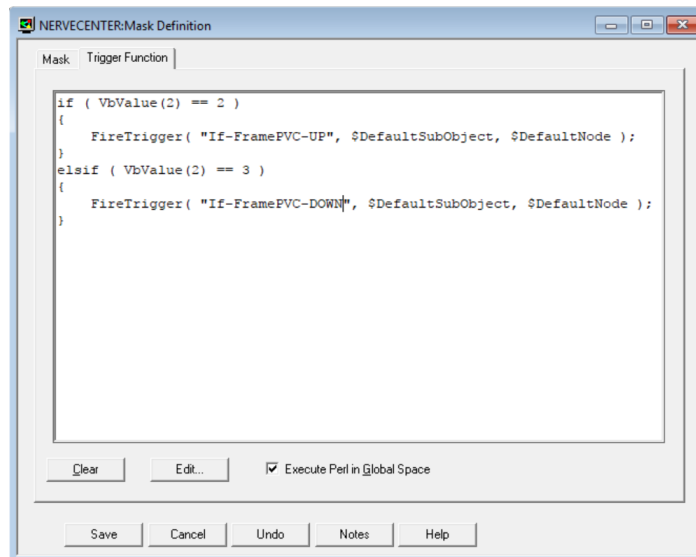


Figure 33: Trigger Function

The notes for this trap mask should look something like this:

Purpose: Detects a trap indicating that a Frame Relay virtual circuit has changed states.

Related alarms: IF-ifFramePVCStatus. This alarm tracks whether the Frame Relay Permanent Virtual Circuit interface is active or inactive.

Vendor information: The trap of interest has an Enterprise of 1.3.6.1.2.1.10.32 (the Frame Relay group) and a Specific trap number of 1. The second variable binding contains the value of frCircuitState, which indicates whether a virtual circuit is invalid (1), active (2), or inactive (3).

Trigger function: If frCircuitState equals 2, the function fires the trigger If-FramePVCUp, and if frCircuitState equals 3, it fires If-FramePVCDown.

Enabling a Trap Mask

For a trap mask to become functional, two conditions must be met:

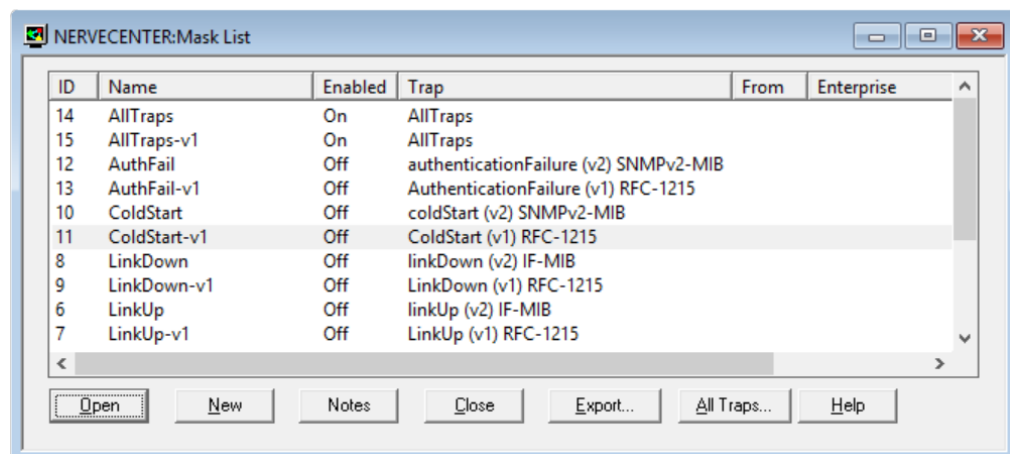
- The trap mask must be enabled.
- There must be an enabled alarm with a *pending* state transition that can be affected by the mask.

This section explains how to enable a trap mask.

TO ENABLE A TRAP MASK

1. From the client's **Admin** menu, choose **Mask List**.

The Mask List window is displayed.

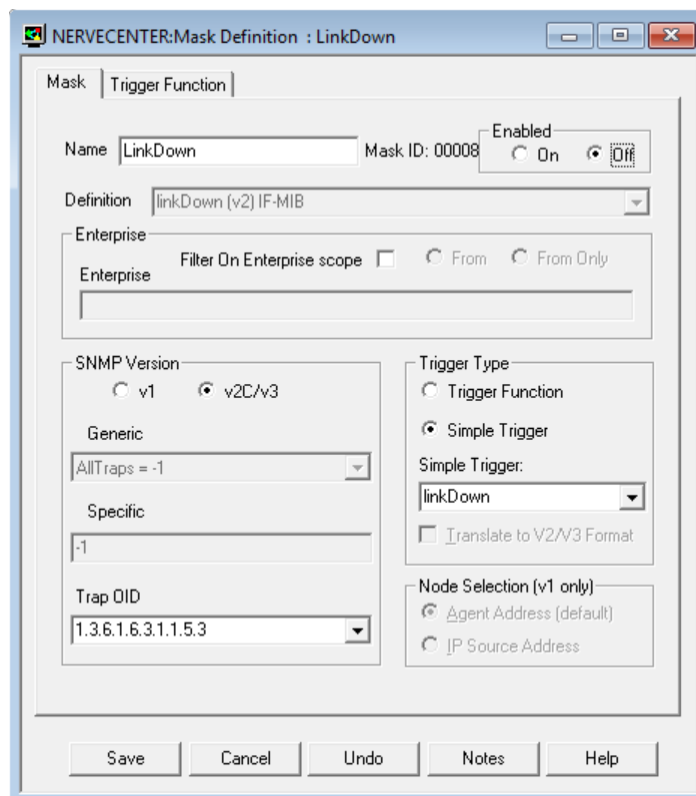


2. Select the mask you want to enable from the list.

The Open button becomes enabled.

3. Select the **Open** button.

The Mask Definition window is displayed and shows the definition of the mask you selected.



4. Select the **On** radio button.

5. Select the **Save** button.

The trap mask is now enabled.

Note: You can also enable a trap mask by selecting a mask in the Mask List window, pressing the right mouse button while your cursor is over the entry for the mask, and choosing **On** from the popup menu.

For the most part, NerveCenter behavior models detect network and system conditions by using polls and trap masks to poll SNMP agents and respond to SNMP traps, respectively. Thus, a behavior model's main source of information is devices running SNMP agents. However, NerveCenter behavior models can obtain data from other sources as well.

For example, a behavior model on one NerveCenter server can receive information from a second NerveCenter server. The second server uses an Inform alarm action to notify the behavior model on the first server of a condition it has detected. This Inform action involves sending what appears to be an SNMP trap to the first server. Actually, the message is not an SNMP trap—it is sent via TCP rather than UDP—but the behavior model receiving it treats it as if it were a trap.

Finally, NerveCenter behavior models can obtain information about network conditions from NerveCenter itself. In particular, when NerveCenter sends an SNMP or ICMP message to a device and the message results in an error (perhaps because the node is unreachable), NerveCenter can notify a behavior model of this condition. NerveCenter does this by using what are called built-in triggers, such as `NODE_UNREACHABLE`, which can cause state transitions in an alarm just as other triggers do. These triggers are necessary because devices that are down or unreachable cannot respond normally to NerveCenter polls, or send SNMP traps to NerveCenter.

Built-In Triggers

When NerveCenter requests a poll, the SNMP GetRequest or the ping that the poll initiates is placed on either NerveCenter's pending SNMP requests list or pending ICMP requests list. NerveCenter waits for a reply from the node or the node's SNMP agent (or from an intervening router). If the node or its SNMP agent sends a non-error reply, then NerveCenter evaluates the poll condition and fires the appropriate trigger.

However, if the node or its SNMP agent does not respond or returns an error—depending upon the circumstances—NerveCenter either retries the request or fires a built-in trigger. Conditions that cause NerveCenter to fire its built-in triggers can be broken down into five categories:

- [SNMP Requests below](#)
- [ICMP Requests on the facing page](#)
- [ICMP Responses on the facing page](#)
- [Matching Errors with Pending SNMP and Ping Requests on page 224](#)
- [Multi-homed Nodes on page 224](#)

Note: NerveCenter uses all uppercase letters to designate built-in trigger names.

For particular information about NerveCenter's built-in triggers, see [Built-In Triggers on page 225](#). For information about the order in which NerveCenter fires built-in triggers, see [Built-in Trigger Firing Sequence on page 223](#).

SNMP Requests

NerveCenter retries SNMP requests as many times as configured or until a reply arrives on the SNMP or ICMP socket that NerveCenter can match to a pending request. (NerveCenter uses the number of retries and retry interval specified on the SNMP tab in the NerveCenter Administrator. Refer to [Specifying SNMP Poll Intervals for NerveCenter](#) in *Managing NerveCenter* for details.)

If the reply is an SNMP error, NerveCenter does not retry the request but returns three built-in triggers with the poll: an **ERROR** trigger, followed by an **SNMP_ERROR** trigger, and then the appropriate SNMP built-in error trigger. (See [Built-In Triggers on page 225](#), for more information.)

If NerveCenter receives no response after the configured number of retries, then NerveCenter fires two built-in triggers: **ERROR**, followed by **SNMP_TIMEOUT**. For more information about the order in which NerveCenter fires built-in triggers, see [Built-in Trigger Firing Sequence on page 223](#).

ICMP Requests

NerveCenter retries ICMP requests as many times as configured or until NerveCenter receives a good, non-error response that it can match to a pending ICMP request. (NerveCenter uses the number of retries and retry interval specified on the SNMP tab in the NerveCenter Administrator. Refer to [Specifying SNMP Poll Intervals for NerveCenter](#) in Managing NerveCenter for details.) If NerveCenter receives no response after the configured number of retries, then NerveCenter fires two built-in triggers: **ERROR**, followed by **ICMP_TIMEOUT**. For more information about the order in which NerveCenter fires built-in triggers, see [Built-in Trigger Firing Sequence on page 223](#).

After the configured number of retries is exceeded, NerveCenter examines the error list, determines which of the matching errors occurred most often, and selects the last packet received from that set. If there is a tie between two or more types of errors, NerveCenter selects the last error packet received. (NerveCenter does not accumulate timeouts. One or more timeouts is counted as only one timeout.)

ICMP Responses

Devices on the network, including the intended device of an SNMP or ICMP request, can respond with an ICMP message. Such messages typically serve to notify of an existing network or configuration issue which prevents normal passage or processing of the initial request. For example, a router within the network may respond with an ICMP message that the intended destination is not reachable per its configuration. Or, a destination host may respond with an ICMP message that the SNMP Agent is not running.

Error details are stored in the attributes of the nl-ping Base Object that NerveCenter includes with each instance of **ICMP_ERROR** that it fires. Using a Perl subroutine or a NerveCenter poll expression, you can extract this data (**Type**, **Code**, **Destination Address** and **Source Address**) to learn more specific information about the ICMP error that occurred.

NerveCenter fires a slightly different sequence of built-in triggers when reporting an ICMP error response as when reporting the timeout of an ICMP request. For ICMP error responses, such as for a net, host or port unreachable condition (where the ICMP fields **Type=3** and **Code=0, 1 or 3**). In this situation, NerveCenter fires an **ERROR** built-in trigger first, followed by an **ICMP_ERROR** trigger, and then finally either a **NET_UNREACHABLE**, **NODE_UNREACHABLE**, or **PORT_UNREACHABLE** built-in trigger.

If the poll times out, NerveCenter fires two built-in triggers: **ERROR**, followed by either an **ICMP_TIMEOUT** or **SNMP_TIMEOUT** trigger.

Multiple Errors Examples

For example, you poll a node with addresses A1, A2, A3, A4 and A5 with the number of retries set to three in the NerveCenter Administrator. The replies are as follows:

Original response = ICMP error E1 from address A1

Response from First retry = ICMP error E1 from address A2

Response from Second retry = no reply within retry interval from address A3

Response from Third retry = ICMP error E2 from address A4

Even though error E2 (third retry) was the last error received, NerveCenter discards it and uses error E1 to produce a response, because it occurred most often. The actual data packet that NerveCenter returns with error E1 is from the first retry, because NerveCenter retains only the last packet for each error code. (The packet from the first retry overwrote the packet from the original response because their error codes matched.)

In this example if any of the ICMP errors contain values for a net, host, or port unreachable condition (where the ICMP fields **Type = 3** and **Code = 0, 1, or 3**), NerveCenter fires an **ERROR** built-in trigger first, followed by an **ICMP_ERROR** trigger, and then finally either a **NET_UNREACHABLE**, **NODE_UNREACHABLE**, or **PORT_UNREACHABLE** built-in trigger. If error E1 is any other ICMP error, then NerveCenter fires two triggers: first, an **ERROR** built-in trigger, followed by an **ICMP_ERROR** built-in trigger that contains data from the first retry packet. For more information about the order in which NerveCenter fires built-in triggers, see [Built-in Trigger Firing Sequence on the facing page](#).

Consider a second example in which the replies are as follows:

Original response = ICMP error E1 from address A1

Response from First retry = ICMP error E2 from address A2

Response from Second retry = ICMP error E3 from address A3

Response from Third retry = no reply within retry interval from address A4

NerveCenter uses error E3 to produce a response because it was the last error received, and no error type occurred more than once. Even though a timeout occurred on the last response, NerveCenter discards it because an error takes precedence over a timeout.

Built-in Trigger Firing Sequence

Table 16 shows the order in which NerveCenter fires built-in triggers.

Table 16: NerveCenter Built-in Trigger Firing Sequence

If the First Trigger Fired is an...	Then the Second Trigger Fired Can Be an...	And the Third Trigger Fired Can Be a...
ERROR	SNMP_ERROR	Specific SNMP built-in trigger
ERROR	ICMP_ERROR	None or, NET_UNREACHABLE, or NODE_UNREACHABLE, or PORT_UNREACHABLE
ERROR	SNMP_TIMEOUT	None
ERROR	ICMP_TIMEOUT	None
ERROR	CANNOT_SEND	None
RESPONSE	Specific non-built-in trigger or None	None
INFORM_CONNECTION_DOWN	None	None
INFORM_CONNECTION_UP	None	None
INFORMS_LOST	None	None
UNKNOWN_ERROR	None	None

Matching Errors with Pending SNMP and Ping Requests

Each poll packet that NerveCenter sends on a socket includes a unique identifier (the IP field **Sequence Number**). When a poll returns ICMP errors within its configured number of retries, NerveCenter collects the error messages that are returned. Each error message includes the sequence number as well as the destination address of the associated node. Certain fields in the ICMP error packet enable NerveCenter to attempt to match SNMP/ICMP error messages with a poll's pending SNMP/ping requests as follows:

- NerveCenter compares a reply on the SNMP socket to its list of pending SNMP requests and attempts to match the reply with the sequence number of an SNMP request. If a match cannot be found with a pending SNMP request, then NerveCenter discards the reply.
- NerveCenter compares a reply on the ICMP socket to its list of pending ICMP requests and attempts to match the reply with the sequence number of an ICMP request. [Table 17](#) summarizes how NerveCenter attempts to match ICMP replies to ICMP pending requests:

Table 17: Matching ICMP Replies with ICMP Requests

Sequence Number Match?	Destination Address In DB?	Action
Yes	Yes	NerveCenter fires the appropriate built-in trigger for the poll.
No	Yes	NerveCenter saves reply to attempt to match with a pending SNMP request.
No	No	NerveCenter discards the reply.

If NerveCenter cannot match the sequence number of an ICMP reply with any pending ICMP requests, but NerveCenter recognizes the destination address, the reply is saved because it might be an error response to an SNMP request for that node; therefore, at regular intervals, NerveCenter compares the destination address of saved ICMP error replies with pending SNMP requests. NerveCenter attempts to match each ICMP reply with the destination address of the oldest pending SNMP request. Only after attempting to match ICMP replies with both pending ICMP and SNMP requests does NerveCenter finally discard the reply when it finds no matches.

Multi-homed Nodes

Polling multi-homed nodes will cause NerveCenter to rotate through the address list for that node in the following manner. If the first address returns an ICMP error response, then NerveCenter flags that address as “down” and will not retry the address until NerveCenter has tried all other addresses for this node.

Upon each retry of a poll, NerveCenter chooses the next IP address to poll. If a node has more addresses than the number of allowable retries, then second or subsequent polls of that node will use the current address if it is “up” or the next un-tried address in the list. If all addresses have been tried, then the “down” addresses will be used again. For an SNMP error, NerveCenter flags the address as “up” because NerveCenter did receive a response from the node's agent.

Built-In Triggers

Table 18 lists all the built-in triggers that NerveCenter can fire.

Note: NerveCenter uses all uppercase letters to designate built-in trigger names.

Table 18: Built-In Triggers

Trigger Name	Meaning
CANNOT_SEND	A local error occurred while NerveCenter was trying to send an SNMP message.
ERROR	An SNMP or ICMP request did not result in a valid response. After firing the ERROR trigger, NerveCenter fires a second trigger that indicates the specific nature of the error.
ICMP_ERROR	Indicates an ICMP error. The ICMP_ERROR trigger contains the ICMP/IP fields from the error message.
ICMP_TIMEOUT	NerveCenter sent an ICMP ping to a node and did not receive a response. This trigger generally indicates that the node in question is down. NerveCenter uses the number of retries and retry interval specified on the SNMP tab in the Administrator. Refer to Specifying SNMP Poll Intervals for NerveCenter in <i>Managing NerveCenter</i> for details.
ICMP_UNKNOWN_ERROR	NerveCenter sent an ICMP ping to a node and received an invalid response. This trigger is no longer used except for the purpose of backward compatibility with version 3.5. We recommend you use it sparingly in the current version.
INFORM_CONNECTION_DOWN	A NerveCenter Inform host connection with paserver is down.
INFORM_CONNECTION_UP	A NerveCenter Inform host connection with paserver was down but is now back up.
INFORMS_LOST	The number of NerveCenter Informs that were unacknowledged and lost, usually while the inform host connection was down.
NET_UNREACHABLE	Indicates that the IP routing layer could not find a route to the network containing the polled node, usually because at least one router was down. This trigger indicates nothing about the status of the node. This trigger can be issued only if you have a router between the workstation running NerveCenter and the polled node.

Trigger Name	Meaning
NODE_UNREACHABLE	Indicates that the IP routing layer could not find a route to the destination node. This trigger indicates nothing about the status of the node. This trigger can be issued only if you have a router between the workstation running NerveCenter and the polled node.
PORT_UNREACHABLE	NerveCenter sent a message to a node, and there was no response from the port to which the message was sent.
RESPONSE	NerveCenter sent an SNMP message and received a valid response from the agent on the destination node.
SNMP_AUTHORIZATIONERR	An SNMPv3 authorization error caused because there is a mismatch between one or all of the rows of vacmAccessTable and the packet. Reasons include: context name mismatch (vacmAccessContextPrefix); security model is not used (vacmAccessSecurityModel); incorrect security level (vacmAccessSecurityLevel); unauthorized to read the MIB view for the SNMP context (vacmAccessReadViewName); unauthorized to write to the MIB view for the SNMP context (vacmAccessWriteViewName); unauthorized to notify the MIB view for the SNMP context (vacmAccessNotifyViewName)
SNMP_BADVALUE	NerveCenter tried to set the value of an attribute in a MIB, but the value it supplied was inappropriate for the attribute. The value may have been of the wrong type, of the wrong length, or invalid for some other reason.
SNMP_DECRYPTION_ERROR	The SNMPv3 engine dropped packets because they could not be decrypted. The 32-bit counter, usmStatsDecryptionErrors , is greater than zero.
SNMP_ENDOFTABLE	NerveCenter fires SNMP_ENDOFTABLE when it finds no more rows while performing an SNMP walk of a MIB table. For example, you could walk IfTable to determine the number of DSO interfaces a node contains.
SNMP_GENERR	A GetRequest, GetNextRequest, or SetRequest failed for some unknown reason (general error).
SNMP_NOSUCHNAME	NerveCenter sent to an SNMP agent a GetRequest, a GetNextRequest, or a SetRequest, and the agent that was contacted was unable to perform the requested operation because: <ul style="list-style-type: none"> ■ The name of the attribute to be read did not match exactly the name of an attribute available for get operations in the relevant MIB view ■ The name of the attribute to be read did not lexicographically precede the name of an attribute available for get operations in the relevant MIB view ■ The attribute to be set was not available for set operations in the relevant MIB view

Trigger Name	Meaning
SNMP_NOT_IN_TIME_WINDOW	The SNMPv3 engine dropped packets because the boots and timeticks sent in the PDU appeared outside of the authoritative SNMP agent's time window. The 32-bit counter, usmStatsNotInTimeWindows , is greater than zero.
SNMP_READONLY	The error readOnly is not defined in RFC 1157. However, some vendors' agents do use this error-status code. As the name implies, the error usually indicates that an agent has received a SetRequest (from NerveCenter, in this case) for an attribute whose access type is read only.
SNMP_TIMEOUT	NerveCenter sent an SNMP message to an agent and did not receive a response. This trigger indicates either that a node's SNMP agent is down or that the node itself is down. NerveCenter uses the number of retries and retry interval specified on the SNMP tab in the Administrator. Refer to Specifying SNMP Poll Intervals for NerveCenter in Managing NerveCenter for details.
SNMP_TOOBIG	An SNMP agent did not respond normally to a GetRequest, GetNextRequest, or SetRequest from NerveCenter because the size of the required GetResponse would have exceeded a local limitation.
SNMP_UNAVAILABLE_CONTEXT	The SNMPv3 engine dropped packets because the context contained in the message was unavailable. The 32-bit counter, snmpUnavailableContexts , is greater than zero.
SNMP_UNKNOWN_CONTEXT	The SNMPv3 engine dropped packets because the context contained in the message was unknown. The 32-bit counter, snmpUnknownContexts , is greater than zero.
SNMP_UNKNOWN_ENGINEID	The SNMPv3 engine dropped packets because they referenced an snmpEngineID that was not known to the SNMPv3 engine. The 32-bit counter, usmStatsUnknownEngineIDs , is greater than zero.
SNMP_UNKNOWN_USERNAME	The SNMPv3 engine dropped packets because they referenced a user that was not known to the SNMPv3 engine. The 32-bit counter, usmStatsUnknownUserNames , is greater than zero.
SNMP_UNSUPPORTED_SEC_LEVEL	The SNMPv3 engine dropped packets because the requested security level is unknown or unavailable. The 32-bit counter, usmStatsUnsupportedSecLevels , is greater than zero.
SNMP_WRONG_DIGEST	The SNMPv3 engine dropped packets because they didn't contain the expected digest value. The 32-bit counter, usmStatsWrongDigests , is greater than zero.
UNKNOWN_ERROR	Some other error occurred.

One additional trigger, `USER_RESET`, is not available from the list of built-in triggers in NerveCenter. NerveCenter fires `USER_RESET` to trigger another state for an existing alarm instance when you reset the alarm instance using the right-click pop-up menu in the Alarm Summary or Aggregate Alarm Summary windows.

An Example Using Built-In Triggers

This section looks at how some of the built-in triggers are used in one of NerveCenter's predefined alarms: `IcmpStatus`. The behavior model of which this alarm is a part repeatedly pings a node to determine its status.

Note: To make the ICMP status behavior model functional, you must turn on the polls `IS_IcmpPoll` and `IS_IcmpFastPoll` and the alarm `IcmpStatus`.

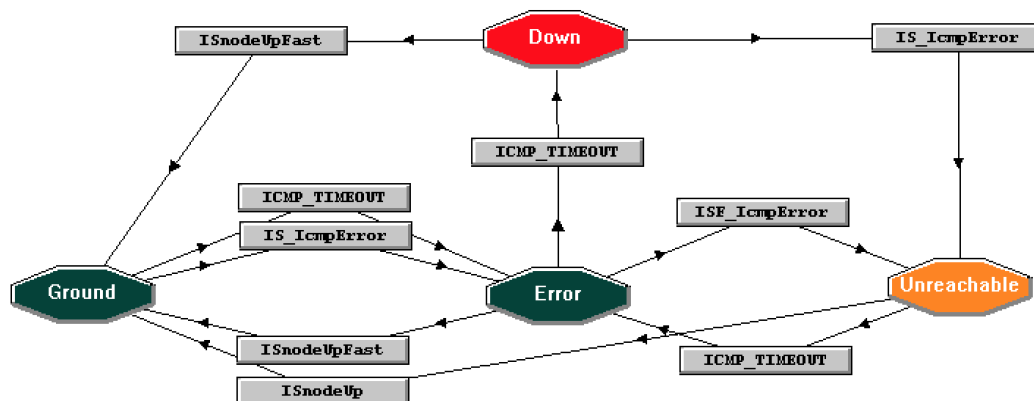


Figure 34: IcmpStatus Alarm

We won't look at every transition in this alarm, but let's look at the alarm's basic design. While the alarm is in the Ground state, NerveCenter is looking for a:

- An error response
(Not an nl-ping-response nor a port unreachable—both indicate that the node is up)
- No response
(ICMP timeout indicated by the built-in trigger `ICMP_TIMEOUT`)

If NerveCenter receives an error response or a timeout, the alarm transitions to the Error state. From the Error state, several things can happen:

- If the node responds to a ping (in which case, either the `ISnodeUp` or `ISnodeUpFast` trigger will be fired by a poll), the alarm transitions back to Ground.
- If the alarm receives another error response, it transitions to the Unreachable state. When the alarm transitions to this state, it puts the node being monitored in a suppressed state.
- If the alarm receives another `ICMP_TIMEOUT` trigger, it transitions to the Down state. On this transition, the alarm puts the node in a suppressed state and sends a message about the problem to a network management platform.

This is only a cursory look at the `IcmpStatus` alarm, but it should give you an idea of how alarms can make use of NerveCenter's built-in triggers.

Another NerveCenter

The section [Using Multiple NerveCenter Servers on page 24](#) introduced the idea of using NerveCenter servers at the various sites within an enterprise to monitor the network conditions at those sites and then to forward important events on to a central NerveCenter server. In this situation, the central server can correlate the events it receives from the remote servers, take appropriate corrective actions, and notify a network management platform of any serious problems it discovers.

Remote servers communicate with the central server using an alarm action called Inform—the same action used to communicate with a network management platform. (For complete information about the Inform alarm action, see the section [Inform on page 286](#).) When a remote server performs this type of Inform action, it sends to the central server what looks like an SNMP trap. This trap's specific trap number is determined by the person who sets up the alarm that initiates the Inform action. The trap also contains a set of variable bindings that include information about the alarm transition that led to the Inform being sent.

Note: These Inform “traps” are not true SNMP traps. Because their receipt by the central server must be guaranteed, they are sent via TCP, not UDP. However, the receiving server processes them as if they were SNMP traps.

The central server handles the traps sent from remote servers just as it handles other traps: by using a trap mask. The only things special about the trap masks you use to receive traps from other NerveCenter servers are that:

- For the trap's enterprise OID, you must supply the OID of the NerveCenter MIB
- For the trap mask's specific trap number, you must supply the specific trap number used in the Inform action

For further information about receiving traps from other NerveCenter servers, see the following sections:

- [Creating a Trap Mask on the next page](#)
- [Variable Bindings for NerveCenter Informs on page 234](#)
- [An Example Trigger Function on page 235](#)

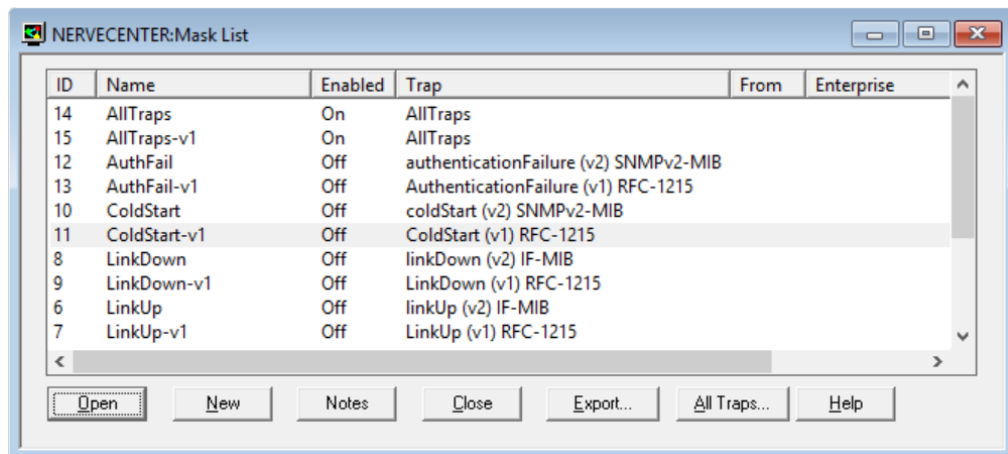
Creating a Trap Mask

This section explains specifically how to create a trap mask designed to receive an Inform trap sent by a remote NerveCenter server. For general information about creating trap masks, see the section [Defining a Trap Mask](#) on page 198.

TO CREATE A TRAP MASK FOR AN INFORM TRAP

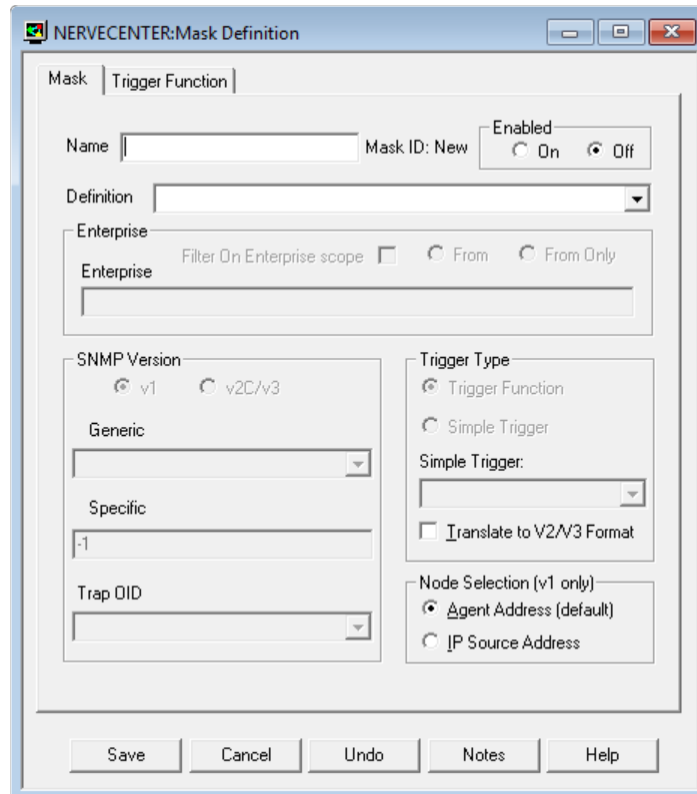
1. From the client's **Admin** menu, choose **Mask List**.

The Mask List window is displayed.



2. Select the **New** button.

The Mask Definition window is displayed.



3. Type a unique name for your trap mask in the **Name** field.

Note: The maximum length for trap mask names is 255 characters.

4. Select **EntSpecific = 6** from the **Generic** drop-down list.
All traps you receive from remote NerveCenter servers are enterprise-specific traps.
5. Select the **From Only** radio button.
6. In the **Enterprise** field, type **1.3.6.1.4.1.78**.
This value will match the value in the Enterprises field of all Inform traps sent from remote NerveCenter servers.

7. Type a specific trap number in the **Specific** field. This value must match the Specific Number used by the remote server's Inform action.

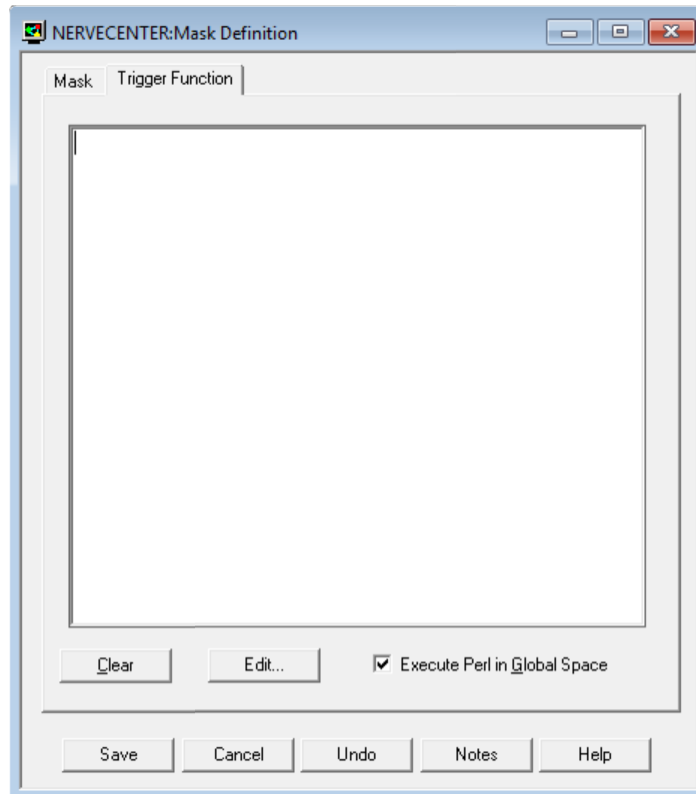
If you want to fire a single trigger if the Generic, Enterprise, and Specific values in the Inform trap match the corresponding values in your trap mask, proceed with [Step 8](#). Otherwise, skip to [Step 11](#).

8. Select the **Simple Trigger** radio button.
9. Type a trigger name in the **Simple Trigger** field, or select a trigger from the **Simple Trigger** drop-down list.
10. Select the **Save** button.

This is the end of the procedure for trap masks that will fire a simple trigger. Be sure to enable your mask when you're ready to use it.

11. Select the **Trigger Function** radio button.

12. Select the **Trigger Function** tab.
The Trigger Function tab is displayed.



13. Enter your trigger function in the text area on the Trigger Function page.
For instructions on writing a trigger function, see the section [Writing a Trigger Function on page 202](#).
14. Select the **Save** button.
Be sure to enable your mask when you're ready to use it.

Variable Bindings for NerveCenter Informs

Depending on how its behavior models are designed, a NerveCenter detecting particular network conditions can send Inform packets to a network management platform or even another NerveCenter Server. Although these Inform packets use TCP/IP, they are similar in content to an SNMP trap, containing trap numbers (generic and specific), an enterprise OID, and a variable-binding list. The lengthy varbinds contains information about the alarm that performed the Inform action, such as the name of alarm, the object the alarm was monitoring, and the names of the origin and destination alarm states.

The network management platform or NerveCenter Server receiving the trap can make use of the information in the variable bindings much the same way it would use variable bindings found in an SNMP trap. For example, the section [An Example Trigger Function on the facing page](#) shows how a NerveCenter server might use some of this information in a trap mask trigger function.

[Table 19](#) explains the contents of this variable-binding list.

Table 19: Inform Trap Variable Bindings

Variable Binding	Value
0	The name of the domain where NerveCenter is running
1	The name of the host machine running the NerveCenter Server
2	The name of the managed node associated with the alarm
3	The base object associated with the alarm (for example, ifEntry for a monitored interface)
4	The base object instance associated with the alarm (for example, 4 for the fourth interface)
5	The name of the subobject. This would include the null string if the alarm is not associated with an alarm.
6	The property group assigned to the node or the subobject
7	The name of the alarm
8	The alarm's property
9	The name of the trigger that caused the alarm transition
10	The state of the alarm before the transition
11	The severity of the state of the alarm prior to the transition
12	The state of the alarm after the transition
13	The severity of the state of the alarm after the transition
14	The maximum severity of all active alarms for the managed node before this alarm transition
15	The maximum severity of all active alarms for the managed node after this alarm transition

Variable Binding	Value
16	The variable bindings in the poll or trap that caused the transition. These variable bindings are formatted as follows: Attribute ncTransitionVarBinds = <i>attribute.instance=value;attribute=value;...</i>
17	The identification number of the alarm instance

An Example Trigger Function

This section explains how you might use an Inform trap's variable bindings in a trigger function. Consider this example: A poll (HighLoad) at a remote site discovers high traffic on an interface and fires the trigger highLoad. This trigger prompts a transition from the medium state to the high state in the alarm ifLoad. (All the objects referred to are actually shipped with NerveCenter.) As shipped, the alarm ifLoad does not perform any actions when the transition from medium to high occurs, but let's say you've added an Inform action that uses the specific number 100005.

The ifLoad alarm (minus the Inform action) also exists at your central site. Therefore, when the Inform trap arrives, you want a trap mask to fire a trigger identical to the one fired at the remote site. In this way, the ifLoad alarm at your central site will stay in sync with the alarm at your remote site.

Here's the trigger function you would have to use in the trap mask at your central site:

```
FireTrigger("highLoad", VbValue(3)..'.'.VbValue(4), VbValue(2));
```

If you recall, the arguments to FireTrigger() are:

- The name of the trigger
- The trigger's subobject (base object plus attribute)
- The trigger's node

The second and third arguments are being retrieved from the list of variable bindings in the Inform trap. For a complete list of the variable bindings included in an Inform trap, see the section [Variable Bindings for NerveCenter Informs on the previous page](#).

Alarms enable you to monitor the state of objects such as interfaces and devices. Figure 35 depicts the role that an alarm typically plays in a behavior model.

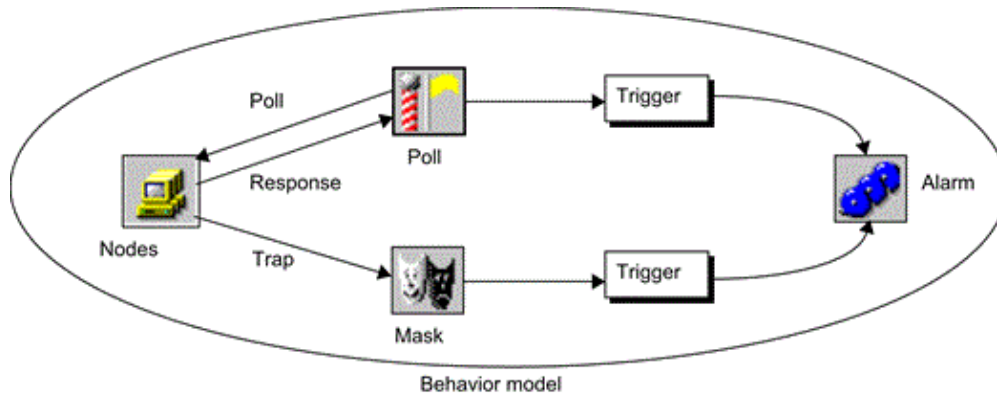


Figure 35: The Role of an Alarm in a Behavior Model

The alarm contains a state transition diagram, and transitions are caused by triggers that are usually generated by polls and trap masks. (Triggers can also be generated by alarms.) When the alarm manager sees a trigger whose key attributes—such as name, subobject, and node—match those of a pending transition in an alarm, the manager causes this transition to take place. Any actions associated with the transition are performed when the transition occurs.

Listing Alarms

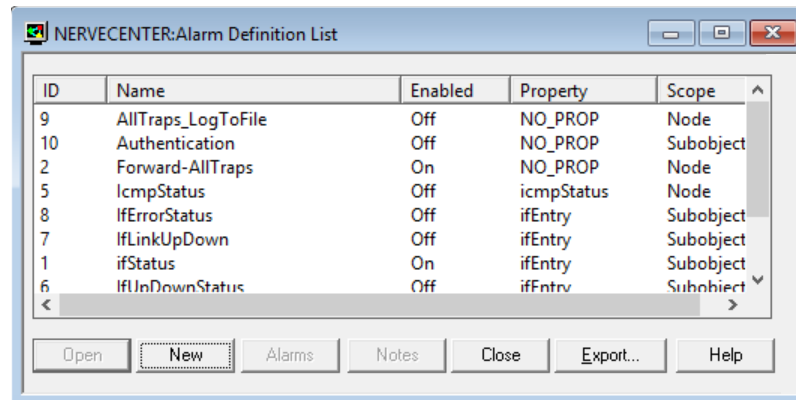
This section explains how to display a list of the alarms currently defined in the NerveCenter database. The section also explains how to view the definition of a particular alarm.

For information on creating a new alarm, see [Defining an Alarm on page 241](#).

TO DISPLAY A LIST OF ALARMS AND THEN DISPLAY A PARTICULAR ALARM'S DEFINITION

1. From the client's **Admin** menu, choose **Alarm Definition List**.

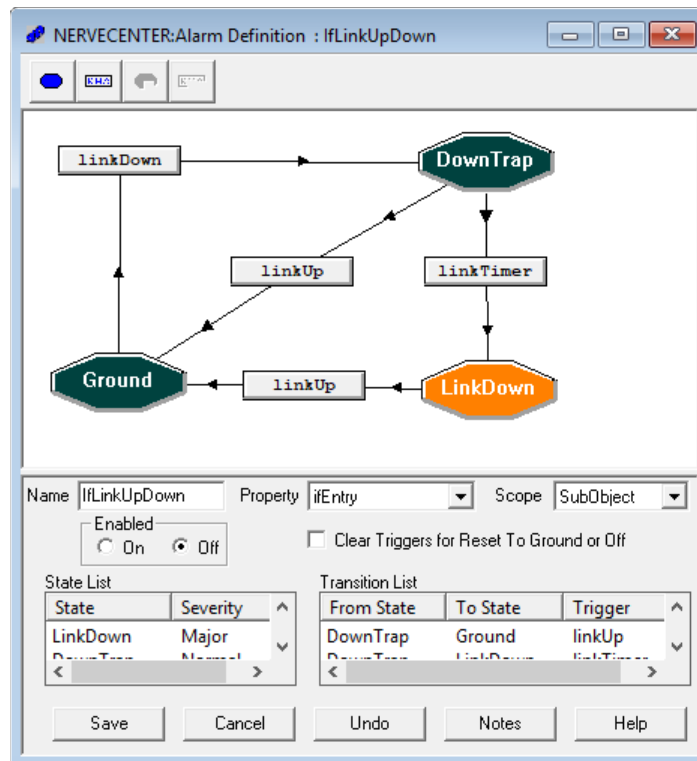
The Alarm Definition List window is displayed.



This window lists all the currently defined NerveCenter alarms and provides a brief definition of each. For each alarm, the window specifies a name and the following information:

- Whether the alarm is currently enabled
 - The alarm's property
 - The alarm's scope
2. Select an alarm from the alarm list.

3. Select the **Open** button
NerveCenter displays the Alarm Definition window.



The alarm defined in this figure is named `ifLinkUpDown`. Each instance of it monitors a single interface (subobject scope) on a device whose property group contains the property `ifEntry`. If NerveCenter receives a generic trap 2 for an interface, an alarm instance is instantiated, and the current state becomes `DownTrap`. If a `linkUp` trap for the same interface arrives within three minutes, the state returns to `Ground`; otherwise, the state becomes `LinkDown`. The state color indicates that `LinkDown` is a state of Major severity.

With a little investigation, you can find out much more about this alarm. For instance, if you right-click a transition, you'll see a pop-up menu that enables you to find out what masks, polls, and alarms can produce the trigger that causes the transition. [Table 20](#) shows what objects can fire the triggers that affect this alarm.

Table 20: Trigger Sources

Transition	Related Trigger Generator
linkDown	Mask: LinkDown
linkUp	Mask: LinkUp
linkTimer	Alarm: IfLinkUpDown

You can also determine what actions will occur on a particular transition. Simply double-click the transition to bring up the Transition Definition dialog. If you perform this task for each transition in this alarm, you'll find that the transition actions in [Table 21](#) have been defined.

Table 21: Transition Actions

Transition	Actions
linkDown (Ground to DownTrap)	Fire the trigger linkTimer on a three minute delay.
linkUp (DownTrap to Ground)	Clear the trigger linkTimer.
linkUp (LinkDown to Ground)	None.
linkTimer (DownTrap to LinkDown)	Inform a network management platform that the interface is down.

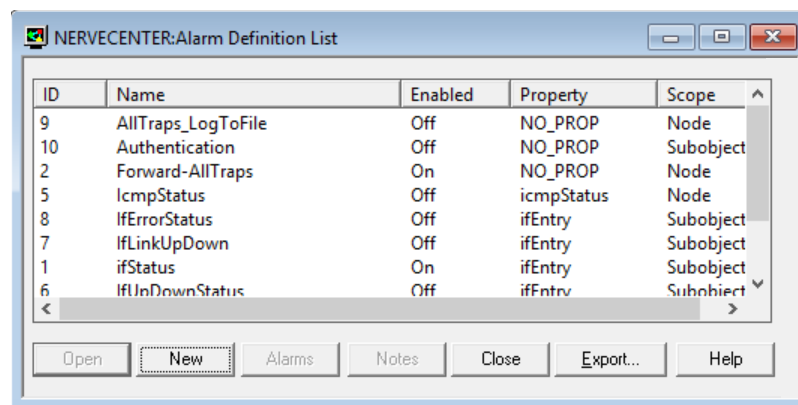
Defining an Alarm

This section provides a high level overview of how to create a new alarm. Because creating an alarm is a fairly involved process, you'll need to consult some additional sections to get all the information you need.

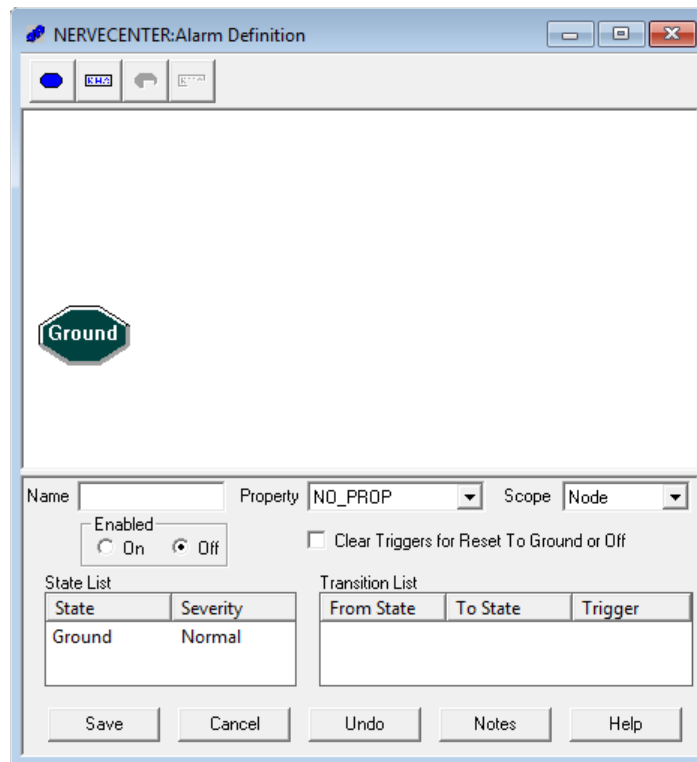
TO DEFINE A NEW ALARM

1. From the client's **Admin** menu, choose **Alarm Definition List**.

The Alarm Definition List window is displayed.



2. Select the **New** button.
The Alarm Definition window appears.



3. Type a unique name for the alarm in the **Name** text field.

Note: The maximum length for alarm names is 255 characters.

4. Select a property from the **Property** list box. Or leave the **Property** set to NO_PROP.
The property you choose helps determine whether a particular trigger can cause an alarm instance to be instantiated or cause a transition in an existing alarm instance. Generally, the alarm's property must match one of the properties in the property group of the node associated with the trigger. The property NO_PROP matches any property.

For complete information regarding the matching rules that determine whether a trigger causes an alarm transition, see the section, [Enabling a Behavior Model's Components on page 371](#).

5. Select a scope from the **Scope** list box.
The options are Enterprise, Instance, Node, and Subobject. An alarm instance with Enterprise scope monitors all nodes managed by the NerveCenter server. An alarm instance with Node scope monitors a single node. A subobject scope alarm monitors a node subcomponent such as an interface (subobject). Instance scope lets you monitor different base objects in a single alarm instance.

For further information on alarm scope, see the section [Alarm Scope on page 244](#).

6. Select the **Clear Triggers for Reset To Ground or Off** checkbox if you want NerveCenter to clear any pending triggers fired by this alarm when the alarm is turned off or manually reset to ground. The alarm might have pending triggers if you associated a Fire Trigger alarm action with this alarm.
 7. Create the alarm's state diagram in the drawing area at the top of the Alarm Definition window. This can be a big step. Before you actually draw the state diagram, you must design it. Your resources for learning how to design an alarms are:
 - This book.
 - [How to Use Alarms](#) in Learning to Create Behavior Models, which includes a tutorial on creating alarms.
 - The predefined alarms that ship with NerveCenter. Looking at these alarms and reading the notes that accompany them should give you some ideas for creating your own alarms.Then, there are the mechanics of creating the state diagram. This subject is covered in the following places:
 - [Defining States on page 245](#)
 - [Defining Transitions on page 249](#)
 - [Alarm Actions on page 269](#) for information about adding actions to alarm transitions
 8. Select the **Save** button to save your alarm.
 9. If you want to enable you alarm now, set the alarm's **Enabled** status to **On**, and then select the **Save** button again.
-

Alarm Scope

NerveCenter alarms can have one of four scopes: subobject, instance, node, or enterprise. A subobject scope alarm monitors a subcomponent of a node, usually an interface (subobject). Instance scope lets you monitor different base objects in a single alarm instance. Node scope monitors activity on a single node, and enterprise scope monitors all managed nodes for a particular behavior.

This is fairly straightforward, but let's look at an example of how alarm scope might affect a particular behavior model. Let's say that you have a model that manages three workstations, each of which has four interfaces.

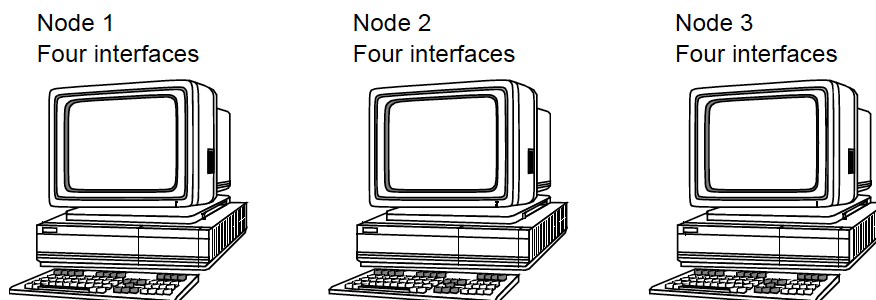


Figure 36: Managed Nodes and Their Interfaces

One component of this behavior model is a poll that checks variables in each workstation's ifEntry table to find interfaces that are experiencing high traffic. This poll can fire a trigger up to twelve times on any poll interval, as shown in [Table 22](#).

Table 22: Triggers Fired by High-Traffic Poll

Node	Subobject
Node 1	ifEntry.1
Node 1	ifEntry.2
Node 1	ifEntry.3
Node 1	ifEntry.4
Node 2	ifEntry.1
Node 2	ifEntry.2
Node 2	ifEntry.3
Node 2	ifEntry.4
Node 3	ifEntry.1
Node 3	ifEntry.2
Node 3	ifEntry.3
Node 3	ifEntry.4

The behavior model also includes the alarm whose state diagram is shown in [Figure 37](#):



Figure 37: High-Traffic Alarm

A beep action is associated with the highLoad transition.

Assuming that you've set the alarm's property properly, you've enabled both the poll and the alarm, and all interfaces are experiencing high traffic, how many beeps will you hear?

The answer depends on your alarm's scope. If the alarm has subobject scope, twelve alarm instances will be created, and you'll hear twelve beeps, one per interface. Similarly, for instance scope, twelve instances will occur and beep. The main difference between subobject and instance scope is that, with instance scope, you could add another transition to the alarm to monitor some base object other than ifEntry.

If the alarm has node scope, three alarm instances will be created, and you'll hear three beeps. Once an alarm instance for a node transitions out of the Ground state—upon receipt of the first highLoad trigger for that node—any subsequent highLoad triggers that refer to that node have no effect. Finally, if the alarm has enterprise scope, only one alarm instance is created, and you'll hear just one beep.

For behavior models that contain just one alarm, choosing an alarm scope is usually simple. Just state the condition you want to be able to detect:

- "I want to be able to detect high traffic on any interface." (Subobject scope)
- "I want to detect several conditions on any interface." (Instance scope)
- "I want to monitor each node on which a particular condition occurs." (Node scope)
- "I want to be notified *the first time* that high traffic occurs on any interface." (Enterprise scope)

Defining States

When you first open the Alarm Definition window, the state-diagram drawing area contains one state. This state is named Ground and is dark green (by default), indicating that the severity of the state is "Normal." This state is unique not only because every alarm must contain it, but because no *active* alarm is ever in this state. The alarm manager instantiates an alarm when it receives a trigger corresponding to a transition from Ground to some other state, and if an alarm instance transitions back to Ground, that instance is deleted.

All of the other states that you want your alarm to track you must create yourself. For example, the author of the predefined alarm IfLoad (interface load) created two nonground states: medium and high.

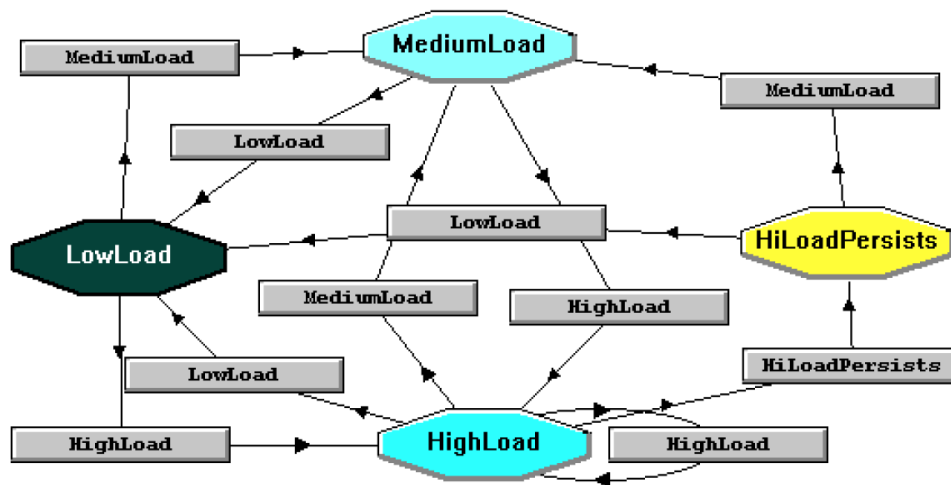


Figure 38: IfLoad Alarm

The medium state is of Medium severity, and the high state is of High severity.

Note: In the alarm in [Figure 38](#), the author has renamed the Ground state “LowLoad.” The Ground state can be renamed and its severity can be changed, but it cannot be deleted.

For instruction on creating new states, resizing state icons, and deleting states, see the following sections:

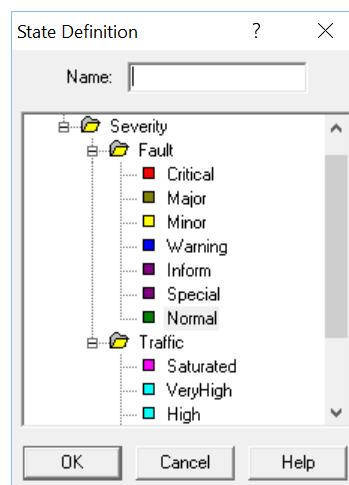
- [Defining a State on the facing page](#)
- [Changing the Size of the State Icons on page 248](#)
- [Deleting a State on page 249](#)

Defining a State

When you add a new state to a state diagram, you must provide two pieces of information about the state: its name and its severity. The name, of course, should indicate the role the state plays in the state diagram. For instance, if a state will indicate that a device is down, you should name it “DeviceDown,” or something similar. The alarm’s severity indicates whether the state represents a fault condition or a traffic condition and how serious the problem is.

TO ADD A STATE TO A STATE DIAGRAM

1. Select the **Add State** button at the top of the Alarm Definition dialog.
The State Definition dialog appears.



2. Type the name of the state in the **Name** text field.

Note: The maximum length for state names is 255 characters.

3. Select a severity from the **Fault** folder or the **Traffic** folder.
4. Select the **OK** button.

The new state appears in the diagram area. Drag the state icon to the spot you want it to occupy in the diagram.

Note: If you don’t move the newly create state, subsequently created states won’t be displayed.

If the state icon’s label won’t fit on the icon, you should resize the state icons in your diagram. For information on how to resize these icons, see the section [Changing the Size of the State Icons on the next page](#).

Changing the Size of the State Icons

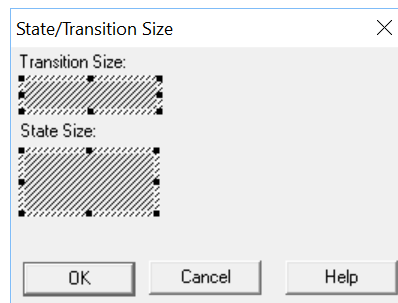
The default size of state icons is fairly small. As a result, the name of a state may not fit on the octagon that represents it. If you encounter this problem, you can change the size of the state icons in your state diagram.

Note: You can't change the size of a single state icon. A resize operation affects all the state icons in the current state diagram.

TO CHANGE THE SIZE OF THE STATE ICONS IN A DIAGRAM

1. Right-click one of the state icons in the diagram, and select **Size** from the pop-up menu that's displayed.

The State/Transition Size window appears.



The rectangle beneath the State Size label indicates the current size of the state icons.

2. Drag the handles on the State Size rectangle to change the width or height of the rectangle.
To accommodate state names that won't fit on icons of the default size, make the rectangle wider.
3. Select the **OK** button.

The width and height of the state icons in your diagram are resized to match the size of the State Size rectangle.

Note: Your state diagram will look better if the names of your states are short.

Deleting a State

If you need to change the name or severity of a state, there's no need to delete the state and create a new one. You can double-click on the icon for the state to bring up the State Definition window and change the state's name or severity there. However, if you've created a state that you no longer need, it's a simple matter to delete it.

TO DELETE A STATE

1. Select the state's icon in your state diagram.
The Remove State button is enabled.
2. Select the **Remove State** button at the top of the Alarm Definition window.
A pop-up dialog asks you whether you're sure you want to remove the state and explains that if you remove a state you also remove all the transitions associated with that state.
3. Select the **Yes** button in the dialog.
The state icon is removed for the state diagram.

Note: You can't delete the Ground state.

Defining Transitions

Once you've created the states for an alarm, you must define the transitions between them. Each transition has these components:

- A origin state.
- A destination state.
- A trigger. This is the trigger that will cause the transition.
- A list of actions that will be performed when the transition occurs. For a full description of each action that can take place upon a transition, see "[Alarm Actions](#)" on page 269

The sections below will lead you through the mechanics of creating a new transition in a state diagram, changing the size of the transition icons in a state diagram, and deleting a transition:

- [Defining a Transition on the next page](#)
- [Associating an Action with a Transition on page 251](#)
- [Changing the Size of Transition Icons on page 252](#)
- [Changing the Color of Transition Lines on page 253](#)
- [Deleting a Transition on page 254](#)

Defining a Transition

When you add a transition to a state diagram, you *must* supply three pieces of information: an origin state, a trigger name, and a destination state. Both of the states must already have been created in the state diagram, and the trigger must already exist as well.

TO CREATE A NEW TRANSITION

1. Select the **Add Transition** button at the top of the Alarm Definition window.
The Transition Definition dialog is displayed.

2. Select an origin state from the **From** drop-down list.
This list contains the names of all the states currently defined in the state diagram, including Ground. If an alarm is in the origin state when the appropriate trigger arrives, it may transition to the destination state.
3. Select a trigger from the **Trigger** drop-down list.
This list contains the names of all the triggers defined in the NerveCenter database. Only a trigger with the name you specify here will be able to cause this transition.
4. Select a destination state from the **To** drop-down list.
5. Select the **OK** button.

A transition is drawn between the source and destination states. This transition consists of a line connecting the source and destination states with arrows pointing in the direction of the destination state, and a rectangular icon on the line labeled with the trigger name. You can drag the rectangular icon, and the line will move with it.

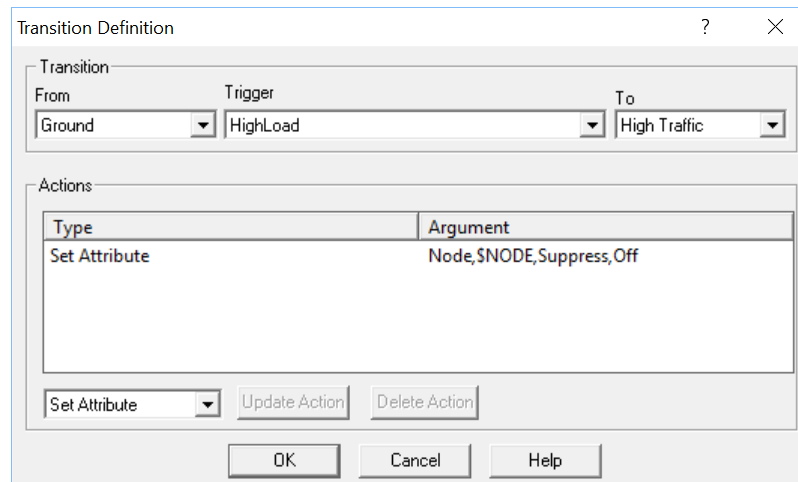
Associating an Action with a Transition

A transition may or may not have alarm actions associated with it. If it has one or more actions associated with it, these actions are performed each time the transition occurs.

You can add actions to an existing transition, or adding the actions can be part of the initial definition of the transition.

TO ADD AN ACTION TO A TRANSITION

1. If you're in the process of creating a new transition, the Transition Definition dialog should already be open. If you want to add an action to an existing transition, double-click the transition's icon. The Transition Definition dialog appears.



2. Select the **New Action** list.
A list of available actions is displayed. See "[Alarm Actions](#)" on page 269 for descriptions.
3. Select an action from the pop-up menu.
If you select the Action Router, Delete Node, or Notes action, the action is added immediately to the Actions list in the Transition Definition window. However, because most actions require you to supply parameters, NerveCenter generally displays an action dialog at this point. The dialog varies from action to action.
4. Fill in the fields in the action dialog.
This step is very dependent on the action you've selected. For details on how to complete this step, see the appropriate section in "[Alarm Actions](#)" on page 269
5. Repeat [Step 2](#) through [Step 4](#) for any additional actions you want to add to the transition.
6. Select the **OK** button in the Transition Definition window.

Changing the Size of Transition Icons

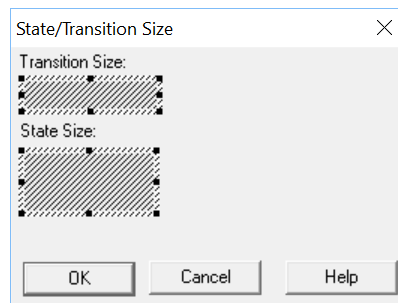
The default size of transition icons is fairly small. As a result, the name of a transition may well not fit on the rectangle that represents the transition. If you encounter this problem, you can change the size of the transition icons in your state diagram.

Note: You can't change the size of a single transition icon. A resize operation affects all the transition icons in the current state diagram.

TO CHANGE THE SIZE OF THE TRANSITION ICONS IN A DIAGRAM

1. Right-click one of the transition icons in the diagram, and select **Size** from the pop-up menu that is displayed.

The State/Transition Size dialog appears.



The rectangle beneath the Transition Size label indicates the current size of the transition icons.

2. Drag the handles on the Transition Size rectangle to change the width or height of the rectangle.
3. Select the **OK** button.

The width and height of the transition icons in your diagram are resized to match the size of the Transition Size rectangle.

Note: Your state diagram will look better if the names of your transitions (triggers) are short.

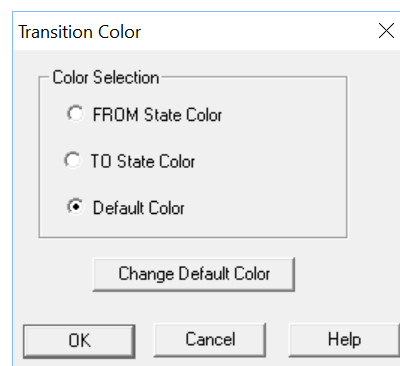
Changing the Color of Transition Lines

Following transition paths with numerous states and transitions can be complicated. Changing the color of the lines connecting states to transitions to make it easier to understand your alarm.

TO CHANGE THE COLOR OF THE TRANSITION LINES IN A DIAGRAM

1. Right-click one of the transition icons in the diagram, and select **Color** from the pop-up menu that is displayed.

The Color Transition dialog appears.



2. Select a color.
 - **FROM State Color** is the color of the state you transition from.
 - **TO State Color** is the color of the state you transition to.
 - **Default Color** is the color you define default to be.

You can define the Default color to any color you want.

- a. Select **Change Default Color**.

The Color window is displayed.

- b. Select the color box containing the color you want to assign to the transition line.
- c. Select the **OK** button in the Color window.

You return to the Transition Color window.

3. Select the **OK** button.

Deleting a Transition

This section explains how to delete a transition from an existing state diagram or one that you're currently drawing.

TO DELETE A TRANSITION

1. Select the transition you want to delete.
2. Select the **Remove Transition** button from the Alarm Definition window.
A dialog appears that asks if you're sure you want to delete the transition.
3. Select the **Yes** button in the dialog.
The transition is deleted from your state diagram.

Bear in mind that an alarm's definition does not actually change until you save the alarm.

Documenting an Alarm

This section explains how to add documentation (notes) to an alarm and what should be covered in that documentation.

- [How to Create Notes for an Alarm on the facing page](#)
- [What to Include in Notes for an Alarm on page 257](#)

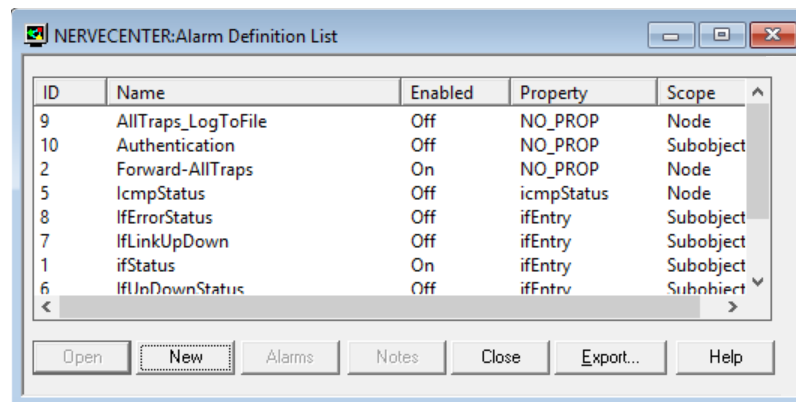
How to Create Notes for an Alarm

You can add notes to an alarm by following the procedure outlined in this subsection.

TO ADD NOTES TO AN ALARM

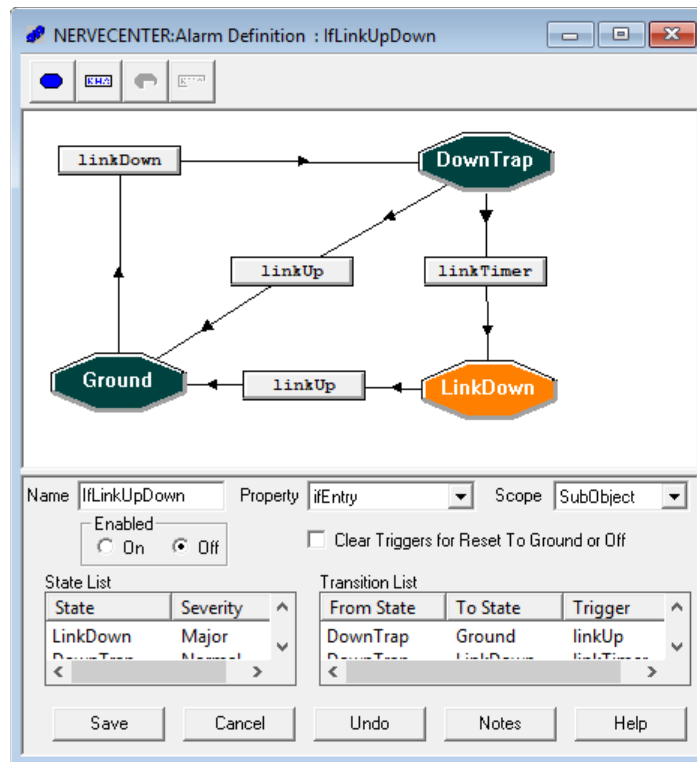
1. From the client's **Admin** menu, choose **Alarm Definition List**.

The Alarm Definition List window is displayed.

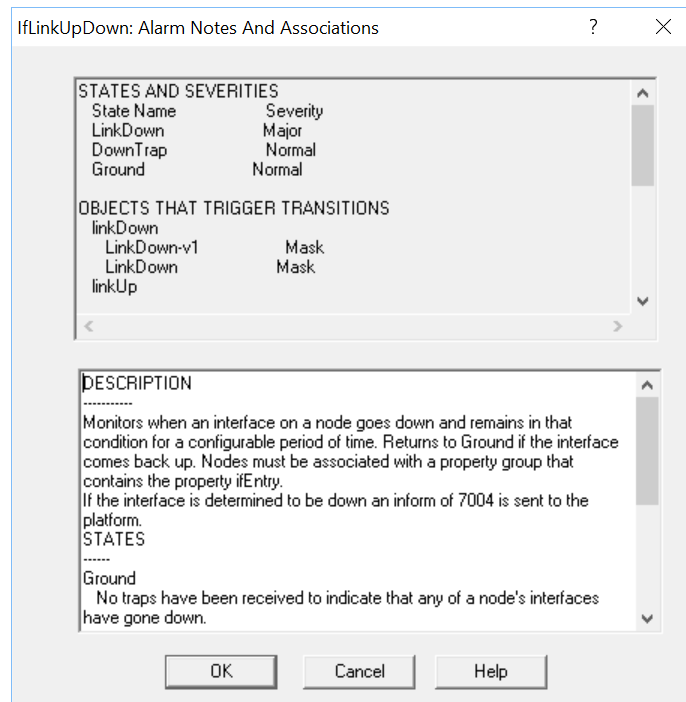


2. Select an alarm to which you want to add a note.
3. Make sure that your alarm is not enabled.

4. Select the **Open** button.
The Alarm Definition window is displayed.



- In the Alarm Definition window, select the **Notes** button.
The Alarm Notes and Associations dialog is displayed.



- Enter your documentation for the alarm by typing in this dialog. See the section [What to Include in Notes for an Alarm](#) below for information on what type of information you should enter here.
- Select the **OK** button at the bottom of the Alarm Notes and Associations dialog to close it.
- Select the **Save** button in the Alarm Definition window.
Your notes are saved to the NerveCenter database. They can be read by anyone who opens the definition for your alarm and selects the Notes button.

What to Include in Notes for an Alarm

The top pane of the Notes and Associations dialog box contains read-only information about the alarm. This data is retrieved from the NerveCenter database and, therefore, may change from time to time when the alarm's definition is modified in the database.

This information includes:

- The alarm's states and severities.
- Objects that trigger the alarm's transitions.
- The alarm's transitions and their associated actions.

The bottom pane contains a general description of the alarm and any useful information. Users with administrator rights can add or edit this field when creating or customizing alarms. Comments should include anything other users might find helpful to know about NerveCenter alarms. Following are some suggestions:

- Define the purpose of the alarm.
- List any reports run against the data logged by the alarm.
- List the property groups that are affected by this alarm—that is, groups that contain the property assigned to the alarm—and include any pertinent information about the nodes assigned to those groups.
- Name any associated behavior model and mention whether customization is required to work with the behavior model.

We recommend that you include the following information in the notes for your alarm:

- Purpose of the alarm
- Brief description of the alarm's states
- Brief description of the alarm's transitions
- List of the objects (polls, masks, and alarms) that fire triggers that affect this alarm
- Description of the actions specified for transitions, especially Fire Trigger and Perl Subroutine actions
- Documentation for any program or script called from a Command action
- Names of any reports run against data logged by the alarm
- Information about other alarms that are part of the same behavior model

For example, let's consider the alarm definition shown in [Figure 39](#).

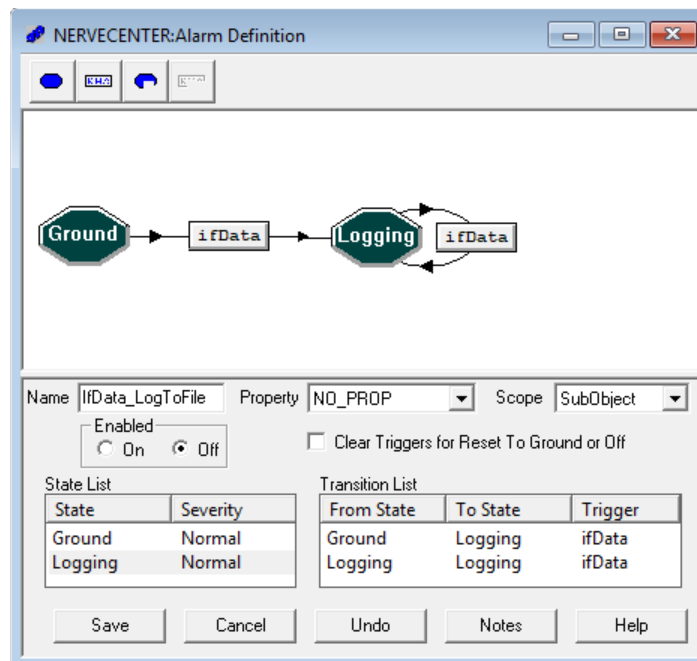


Figure 39: IfData_LogToFile Alarm

The notes for this alarm should look something like this:

```
Purpose: Logs interface data to the log file ifdata.log.  
States: Ground (Normal), Logging (Normal)  
Transitions: ifData (Ground to Logging), ifData (Logging to  
Logging)  
Associated poll: IfData fires the ifData trigger if it is able to  
retrieve information about an interface from a node's interface  
table.  
Actions: ifData (Ground to Logging) - Log to File ifdata.log  
Enabled Verbose  
ifData (Logging to Logging) - Log to File ifdata.log Enabled  
Verbose
```

Enabling an Alarm

For an alarm to become functional, several conditions must be met:

- The alarm must be enabled.
- The alarm must receive a trigger that corresponds to one of the alarm's transitions out of the Ground state.
- The alarm's property must be in the property group of the node associated with the trigger.

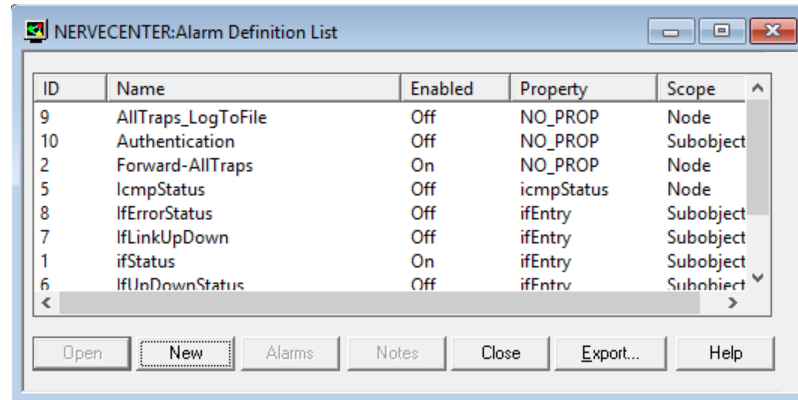
This section explains how to enable an alarm.

Note: If you later turn an alarm off or reset the alarm to ground, any pending triggers fired by that alarm are cleared if the **Clear Triggers for Reset To Ground or Off** checkbox is checked in the alarm's definition window.

TO ENABLE AN ALARM

1. From the client's **Admin** menu, choose **Alarm Definition List**.

The Alarm Definition List window is displayed.

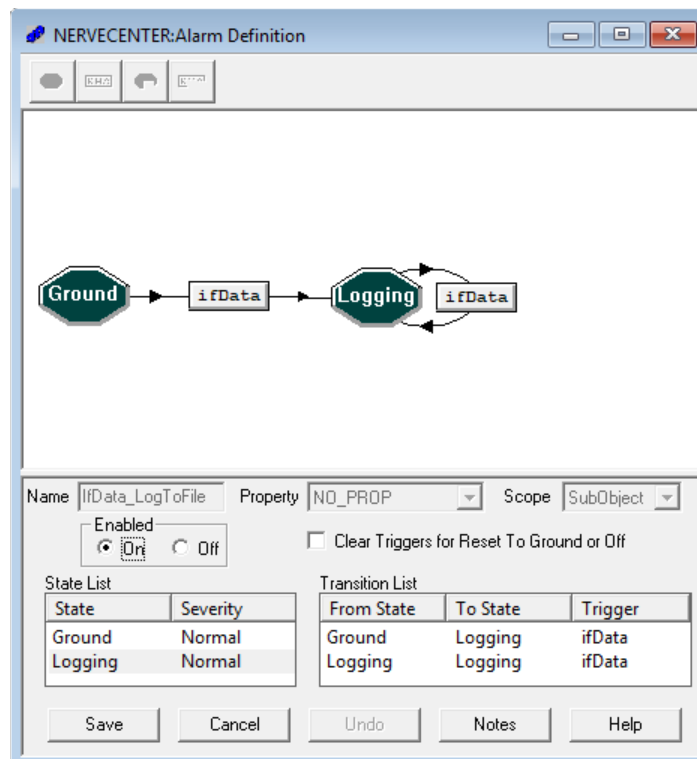


2. Select the alarm you want to enable from the list.

The Open button becomes enabled.

3. Select the **Open** button.

The Alarm Definition window is displayed and shows the definition of the alarm you selected.



4. Select the **On** radio button in the **Enabled** frame.
5. Select the **Save** button.

The alarm is now enabled.

Note: You can also enable an alarm by selecting the alarm in the Alarm Definition List window, right-clicking the entry for the alarm, and choosing **On** from the popup menu.

Correlation Expressions

NerveCenter provides an additional method for Alarm Definition creation, the Correlation Expression window. Correlation expressions allow the definition of alarm diagrams based on Boolean expressions. The correlation expressions do not apply in every situation, but in cases where multiple combinations of events need to be detected and acted upon, the correlation expressions save tremendous amounts of time, both in alarm diagram designing and building.

To build a correlation expression, first create the necessary trap masks and poll conditions to fire the desired triggers. Once the triggers have been created, the Correlation Expression Editor can be used to create the expression.

There are three main components of the correlation expression. First, the Boolean expression is created using and, or, parenthesis and triggers that are already existent. Second, the correlation reset period determines the time limit in which the entire expression must become true once a portion has been detected. Third, the correlation action must be specified, directing NerveCenter to act when the expression becomes true.

For example take the sample alarm in [Figure 40](#).

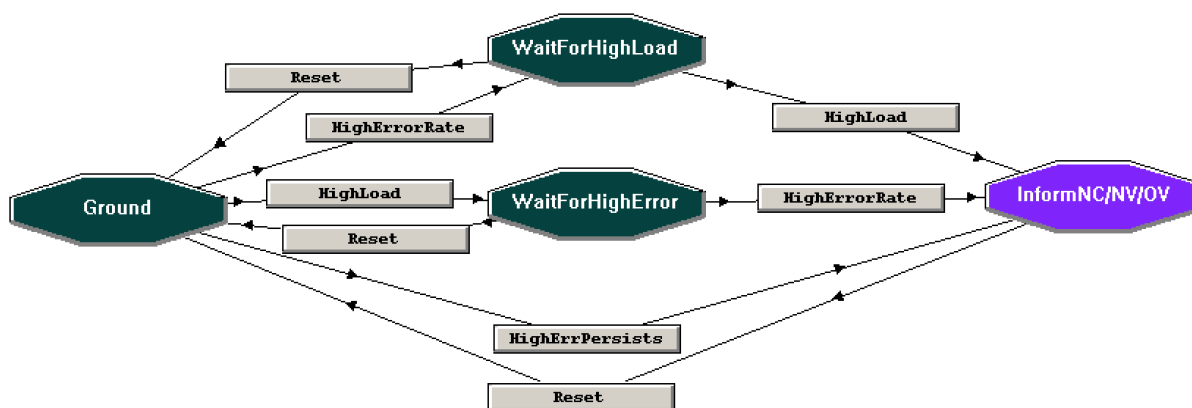


Figure 40: Sample Alarm: Error Rate Alarm Created with the Alarm Definition Window

In this alarm, you want an inform to be sent if you receive HighErrorRate and HighLoad triggers or if you receive a HighErrorPersists trigger. The alarm will reset to Ground if the alarm is not completed within the time period specified by the transition Reset. Creating this model takes several steps. You need to create three states and eight transitions. Three of those transitions require you to add the same action, send inform. The idea behind this model, however, can be expressed simply with a boolean expression:

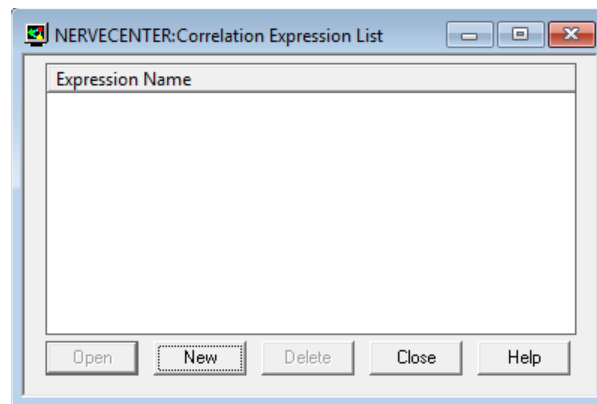
If (HighLoad AND HighErrorRate) OR HighErrPersists, then Inform IC/NV/OV

Correlation expressions allow you to create simple alarms quickly.

Note: After a correlation expression reaches the final state, the Alarm reverts to Ground.

TO CREATE A CORRELATION EXPRESSION

1. From the client's **Admin** menu, choose **Correlation Expression List**.
The Correlation Expression List Window opens.



2. Select the **New** button.

The Edit Correlation Expression window opens.

3. In the **Expression Name** field enter a name for the expression.

Note: The maximum length for correlation names is 255 characters.

4. Enter the **Reset Period** (must be greater than 0) and select a time unit (seconds, minutes or hours) from the drop-down menu.

The correlation reset period is the time in which the entire alarm must complete before the alarm resets. This counter starts when the first trigger occurs. The counter does not restart when a second trigger occurs.

The time period must be greater than zero. You can choose between seconds, minutes or hours.

5. Enter a correlation expression.

You enter information in the Correlation Expression field by using the buttons below the field.

- To add a trigger:
 - a. Select a trigger from the **Trigger to Add** drop down list.
 - b. Select **Add Trigger**.
- To add a Boolean operator, select the **AND** or **OR** button.

Note: The AND operator has precedence over the OR operator. For example, **x or y and z** is the same as **x or (y and z)**.

- To add a parenthesis, select the (or) button.
The close parenthesis) button is not active until there is an open parenthesis (in the correlation expression.
- To delete the previous element of the correlation expression, select the **Del** button.

6. Add Correlation Actions.

- Select the **New Action** list.
A list of available actions is displayed. See "[Alarm Actions](#)" on page 269 for descriptions.
- c. Select an action from the pop-up menu.
If you select the Action Router, Delete Node, or Notes action, the action is added immediately to the Actions list in the Edit Correlation Expression window. However, because most actions require you to supply parameters, NerveCenter generally displays an action dialog at this point. The dialog varies from action to action.
- d. Fill in the fields in the action dialog.
This step is dependent on the action you've selected. For details on how to complete this step, see the appropriate section in "[Alarm Actions](#)" on page 269.
You can edit these selections later by selecting the **Update Action** button.
- e. Repeat [Step b](#) through [Step d](#) for any additional actions you want to add to the correlation expression.

To delete an action, select it from the Correlation Actions list and click **Delete Action**.

7. Select **Save**.

Note: The **Save** and **Create Alarm** buttons are not enabled until: 1) you give the correlation expression a name; 2) you set the Reset Period to a number other than zero; 3) the correlation expression is valid (for example, all open parenthesis are closed); 4) you select at least one Correlation Action

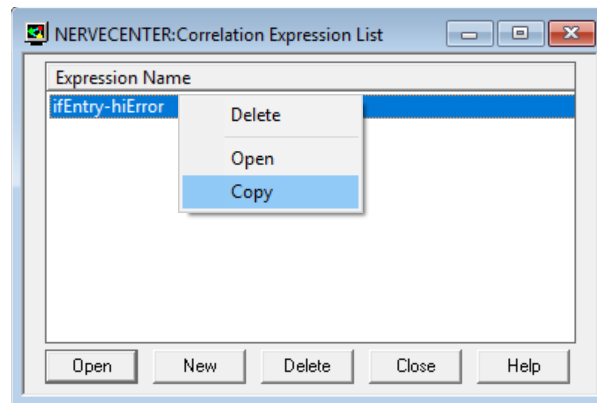
This saves the correlation expression.

After creating a correlation expression, you can use it as a building block to create alarms.

Note: You do not have to save a correlation expression to create an Alarm. As long as the correlation expression has a name, a reset period, a valid expression and an action, you can create an Alarm from the expression.

TO COPY A CORRELATION EXPRESSION

1. From the client's **Admin** menu, choose **Correlation Expression List**.
The Correlation Expression List Window opens.



2. Select a correlation expression and right-click.
3. Select **Copy** from the pop-up menu.
The Edit Correlation Expression window opens.
4. In the **Expression Name** field enter a new name for the expression.

Note: The maximum length for correlation names is 255 characters.

5. Select **Save**.

TO CREATE AN ALARM FROM A CORRELATION EXPRESSION

1. From the Edit Correlation Expression window, click **Create Alarm**.
The Create Alarm using Correlation Expression window opens.

2. In the Alarm Name field, enter a name for the alarm.

Note: The maximum length for alarm names is 255 characters.

3. Select a property from the **Property** list box or leave the **Property** set to NO_PROP.
The property you choose helps determine whether a particular trigger can cause an alarm instance to be instantiated or cause a transition in an existing alarm instance. Generally, the alarm's property must match one of the properties in the property group of the node associated with the trigger. The property NO_PROP matches any property.
4. Select a scope from the **Scope** list box.
The options are Enterprise, Instance, Node, and Subobject. Briefly, an alarm instance with Enterprise scope monitors all the nodes managed by the NerveCenter server. An alarm instance with Node scope monitors a single node. A subobject scope alarm monitors a subcomponent of a node, usually an interface (subobject). Instance scope lets you monitor different base objects in a single alarm instance.

For further information on alarm scope, see [Alarm Scope on page 244](#).

5. Select the **Clear Triggers for Reset To Ground or Off** checkbox if you want NerveCenter to clear any pending triggers fired by this alarm when the alarm is turned off or manually reset to ground. The alarm might have pending triggers if you associated a Fire Trigger alarm action with this alarm.

6. If you want to enable you alarm now, set the alarm's **Enabled** status to **On**.
7. Select **Save Alarm**.

When you save the alarm, you can now access it through the Alarm Definition List and edit it as any other alarm. For details on using the Alarm Definition window, see [Defining Transitions on page 249](#).

Figure 41 shows the correlation expression that creates the alarm shown in Figure 40 on page 261. Figure 42 shows the alarm generated with the Error Rate correlation expression.

Figure 41: Error Rate Correlation Expression

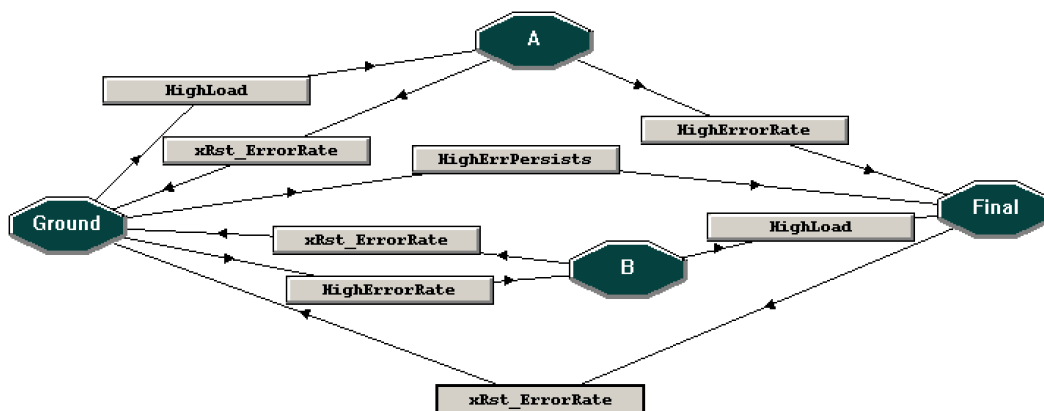
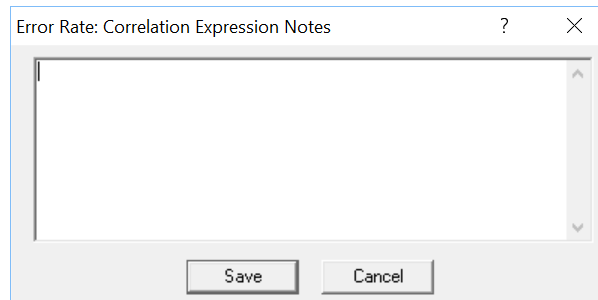


Figure 42: Error Rate Alarm Generated from the Error Rate Correlation Expression

TO ADD NOTES TO A CORRELATION EXPRESSION

1. From the Correlation Expression window, select **Notes**.
The Correlation Expression Notes dialog box displays.



2. Enter your comments.
3. Select **Save** to close the Notes dialog box.
A dialog box asking **Are you sure?** displays.
4. Select **Yes**.
5. Click **Save** in the Correlation Expression window to save the notes.

Note: These notes document the correlation expression. They are not copied over to any alarm created by a correlation expression.

When you create an alarm, you can specify that one or more alarm actions take place on any alarm transition. These actions fall into two categories: those that affect how the alarm works and those that perform some type of corrective action. An example of the first type of action is the Fire Trigger action. This action (as its name implies) fires a trigger that can cause a transition in its own or another alarm. An example of the second type of action is the Command action, which enables you to run any script or executable when a transition occurs.

Note: NerveCenter alarm actions are asynchronous. Alarm actions do not execute in the order that you specify them—actions can fire in any order. Therefore, action2 should not be dependant on action1, for example.

The only exception is the Clear Trigger action; when you include a Clear Trigger action with other alarm actions, the Clear Trigger action is always performed first. This prevents the possibility of a trigger being fired and then cleared during the same transition.

Action Router

Normally, when an alarm transition occurs, the actions associated with that transition are performed automatically. However, it's possible to specify that one or more actions be performed conditionally. To define this type of behavior, you must:

- Add the Action Router action to the appropriate alarm transition. (This section explains how to perform this task.)
- Use the Action Router's rule composer to define the conditions under which you want the Action Router to perform one or more actions and the actions to be taken under those conditions. These conditions can be specified using any Perl expression that evaluates to true or false. However, NerveCenter provides a large set of variables for use in these conditions. These variables enable you to set up conditions based (among other things) on:
 - The name of the alarm that underwent the transition
 - The day of the week
 - The name of the node being monitored
 - The property group associated with the node being monitored
 - The severity of the transition's destination state

- The time of day
- The name of the trigger that caused the transition

In addition, the actions that can be associated with a set of conditions can be selected from almost all the actions that can be performed during an alarm transition. For complete information about using the rule composer, see "[Performing Actions Conditionally \(Action Router\)](#)" on page 315

Once you've done this setup, when the transition with the Action Router action takes place, the Action Router process will receive information about the transition. The Action Router will then evaluate all of its rules to determine any of them are satisfied. If a rule is satisfied, the Action Router performs all of the actions associated with that rule. For example, if you've set up a rule that tells the Action Router to page an administrator if a transition's destination state is of Critical severity, the Action Router will check the transition's destination state and page an administrator if that state is Critical.

TO ADD THE ACTION ROUTER ACTION TO AN ALARM TRANSITION

1. In the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Action Router** from the pop-up menu.
The new action appears in the Actions list in the Transition definition window.
3. Select the **OK** button in the Transition Definition window.
4. Select **Save** in the Alarm Definition window.

Alarm Counter

Suppose that you want to write an alarm to detect more than five authentication-failure traps from a node within five minutes. A possible state diagram for this problem is shown in Figure 43.

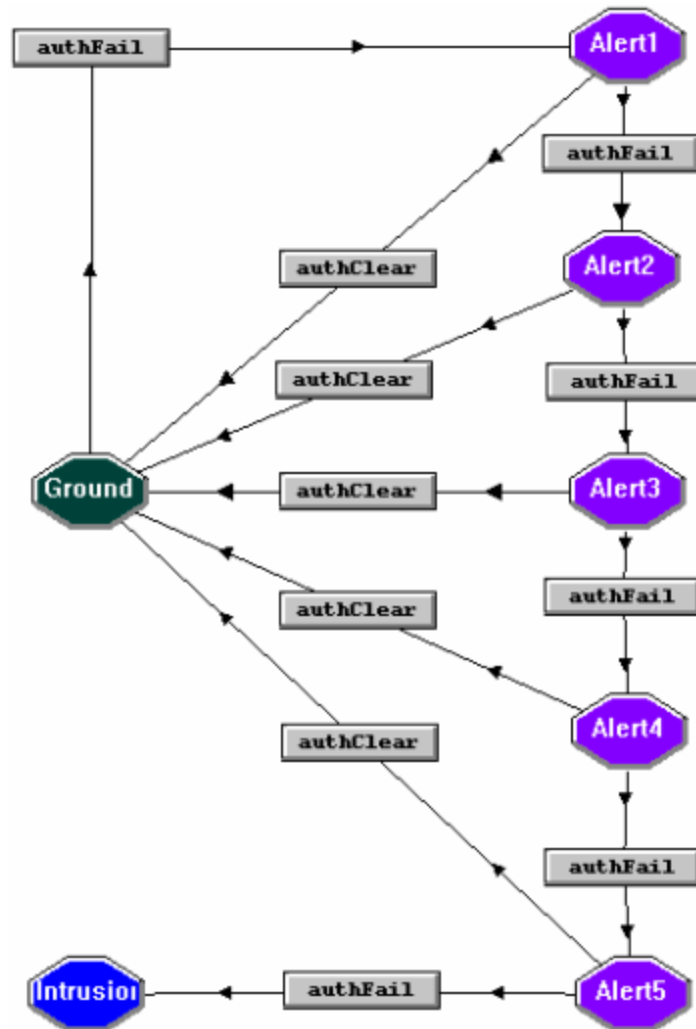


Figure 43: First Solution to Authentication-Failure Problem

Presumably, the trigger `authFail` is fired by a trap mask that detects generic authentication-failure traps. Also, on the transition from `Ground` to `Alert1`, the trigger `authClear` is fired on a five minute delay. This trigger is cleared on the transition from `Alert5` to `Intrusion`.

With seven states, this diagram doesn't look too bad. But what if you had been asked to write an alarm that detected more than twenty authentication failures? Clearly, a better approach is needed.

The NerveCenter feature that you can use to simplify this type of state diagram is the Alarm Counter alarm action. This action enables you to loop in an alert state until you're ready to move to the Intrusion state. Thus, a revised state diagram might look like Figure 44:

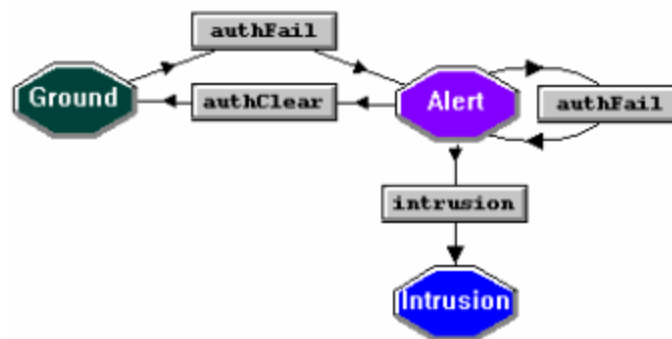


Figure 44: Solution Using the Alarm Counter Action

The firing and clearing of the authClear trigger are handled as they were in the previous example. The new actions in this state diagram are Alarm Counter actions on both the transition from Alert to Alert and the transition from Alert to Ground.

The Alarm Counter action associated with the circular transition from Alert to Alert:

- Creates a counter variable if it does not already exist.
- Increments the counter. (The initial value of the counter is zero.)
- Checks to see whether the value of the counter is 5. (The test is for 5 instead of 6 because one authorization failure has to occur for the alarm to reach the Alert state.)
- Fires the trigger intrusion if the value of the counter is greater than 4.

The Alarm Counter action associated with the transition from Alert to Ground:

- Creates the counter if it does not already exist.
- Sets the value of the counter to 0.

This example shows both of the main uses of the Alarm Counter action: to set up a loop in which a trigger is fired when the counter reaches a certain value and to set or reset the value of the counter.

Note: You can use the "[Counter\(\) Function](#)" on page 299 in a Perl subroutine or Action Router rule to get the value of a counter associated with a particular transition. For more information, see [Counter\(\) Function on page 299](#).

You can also use the `NC::AlarmCounters` Perl object in Perl subroutines. However, the `NC::AlarmCounters` object is completely separate from the `Counter()` function and does not share data with the `Counter()` function. For more details about the `NC::AlarmCounters`, see the NerveCenter Release Notes.

TO CREATE AN ALARM COUNTER

1. In the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Alarm Counter** from the pop-up menu.
The Alarm Counter Action dialog is displayed.

3. Type a counter name in the **Counter Name** text field, or select a counter name from the **Counter Name** drop-down list.
The drop-down list will contain values only if another transition in the same alarm has already defined an alarm counter.

The scope of the alarm counter name is the alarm instance in which the counter is created.
4. To set up a loop—that is, you want to fire a trigger after a transition has occurred a certain number of times—perform these steps:
 - a. Select either the **Increment** or **Decrement** radio button.
Obviously, this choice determines whether the counter will be incremented or decremented when the Alarm Counter action is performed. Normally, you increment a counter because the counter is initialized to 0. However, it is possible to set the counter to a nonzero value in one Alarm Counter action and then to decrement it in another.

The counter is incremented or decremented before it is used in any comparison.
 - b. Type an integer in the **when counter equals** field.
The Alarm Counter action can fire a trigger when the counter equals this value.
 - c. Type the name of a new trigger in the **Fire Trigger** field, or select an existing trigger from the **Fire Trigger** drop-down list.
If you do not enter a trigger name, any value you enter in the “when counter equals” field is lost when you save the alarm.

5. To set or reset the value of a counter, perform these steps:
 - a. Check the **Set Counter** checkbox.
 - b. Enter an integer in the **Value** field.

The counter will be set to this value when the Alarm Counter action occurs.

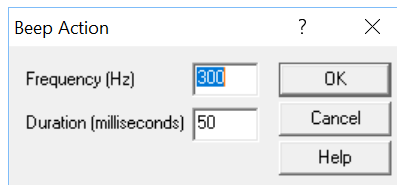
6. Select the **OK** button in the Alarm Counter Action dialog.
7. Select the **OK** button in the Transition Definition window.
8. Select the **Save** button in the Alarm Definition window.

Beep

If you add the Beep alarm action to a transition, NerveCenter sends an audible alarm to all of the clients connected to the server when that transition occurs. This is one method of notifying network administrators of a condition that requires their attention.

TO ADD A BEEP ALARM ACTION TO A TRANSITION

1. In the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Beep** from the pop-up menu.
The Beep Action dialog is displayed.



3. Type a value in the **Frequency** field, or leave the default value of 300.
This value specifies the beep's frequency in hertz.
4. Type a value in the **Duration** field, or leave the default value of 50.
This value specifies the beep's duration in milliseconds.
5. Select the **OK** button in the Beep Action dialog.
6. Select the **OK** button in the Transition Definition window.
7. Select the **Save** button in the Alarm Definition window.

Clear Trigger

When you define a Fire Trigger alarm action, you can use a delay to determine when the trigger actually fires. (For details about the Fire Trigger action, see the section [Fire Trigger on page 280](#).) After a Fire Trigger action has been initiated, but before the delay has elapsed, you can cancel the firing of the trigger using the Clear Trigger action. A Clear Trigger action cancels any pending triggers of a specified name that have been queued by its own alarm instance.

When you include a Clear Trigger action with other alarm actions, the Clear Trigger action is always performed first. This prevents the possibility of a trigger being fired and then cleared during the same transition.

A good example of the use of Fire Trigger and Clear Trigger is the predefined alarm `IfLinkUpDown`.

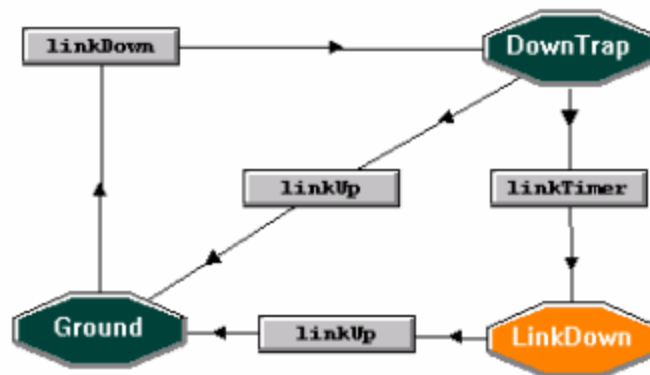
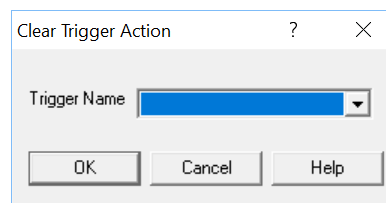


Figure 45: IfLinkUpDown Alarm

This alarm is designed to transition from Ground to Down Trap upon the receipt of a linkDown trigger. When this transition occurs, a Fire Trigger action fires the trigger linkTimer on a three-minute delay. If a linkUp trap arrives within three minutes, the linkUp transition occurs, and a Clear Trigger action clears the linkTimer trigger. Otherwise, the linkTimer trigger is fired, and the alarm transitions to the LinkDown state.

TO ADD A CLEAR TRIGGER ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select the **Clear Trigger** action.
The Clear Trigger Action dialog is displayed.



3. Type the name of the trigger you want to clear in the **Trigger Name** field, or select it from the **Trigger Name** drop-down list.
Pending triggers of this name will be cleared only in the alarm instance that invokes the Clear Trigger action.
4. Select the **OK** button in the Clear Trigger Action dialog.
5. Select the **OK** button in the Transition Definition window.
6. Select the **Save** button in the Alarm definition window.

Command

The Command alarm action enables you to execute any command or script when a particular alarm transition occurs. An example of an alarm that uses this action is the predefined alarm IPSweep.

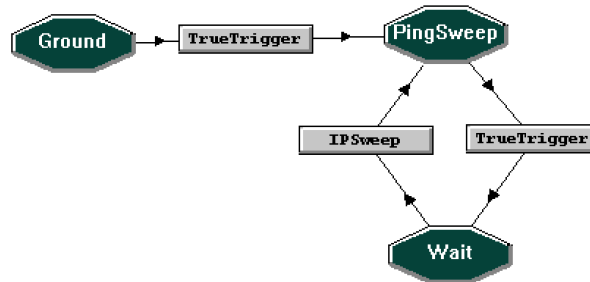
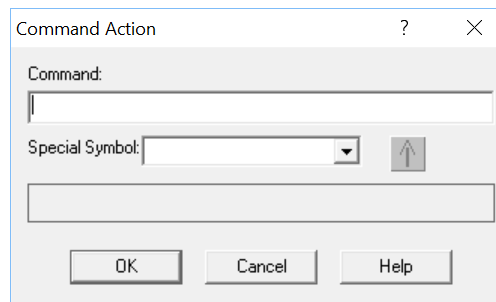


Figure 46: IPSweep Alarm

When the IPSweep transition occurs, this alarm executes a program called ipsweep. This is the program that actually discovers the devices on the subnets you're managing.

TO ADD A COMMAND ACTION TO A TRANSITION

1. In the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Command** from the pop-up menu.
The Command Action dialog is displayed.



3. Type the command to be executed in the **Command** field.
On Windows systems, the command can be any .exe, .bat, or .cmd file you can invoke from the command line. You can omit the command suffix because the operating system will locate the appropriate file. On UNIX systems, the command can be any executable binary or script file that you can invoke from a shell.

4. Enter any parameters that the command requires after the command.

Note: The command plus its parameters can be up to 2020 characters in length. If you exceed this length, the error “Command line too long” is written to the event or system log.

If the parameters are constants, you can simply type them in the Command field following the command name. However, if they will vary from alarm instance to alarm instance (and NerveCenter maintains the information you need in one of its variables), you can use the Special Symbol drop-down list and the button beside it to enter the parameters. For more information, see [NerveCenter Variables on page 301](#).

To enter a variable in your command:

- a. Place your cursor at the appropriate spot in the **Command** field.
 - b. Select a variable from the **Special Symbol** drop-down list.
 - c. Select the button to the right of the Special Symbol field.
5. Select the **OK** button in the Command Action dialog.
 6. Select the **OK** button in the Transition Definition window.
 7. Select the **Save** button in the Alarm Definition window.

Delete Node

The Delete Node action deletes the node being monitored by the current alarm instance from the NerveCenter database.

An example of using Delete Node might be to remove a node from the NerveCenter database that does not respond to a ping for five minutes after an alarm transitions to a down state.

TO ADD A DELETE NODE ACTION TO A TRANSITION

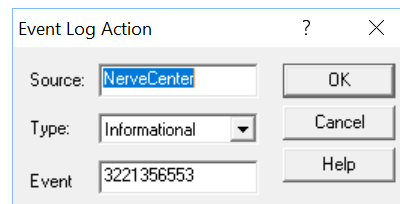
1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
 2. Select **Delete Node** from the pop-up menu.
The Delete Node action is added to the Actions list in the Transition Definition window.
 3. Select the **OK** button in the Transition Definition window.
 4. Select the **Save** button in the Alarm Definition window.
-

EventLog

The EventLog alarm action writes information about an alarm transition to the UNIX system log file. On Solaris the system log file is `/var/adm/messages`; on Linux the system log file is `/var/log/messages`.

TO ADD AN EVENTLOG ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **EventLog** from the pop-up menu.
The Event Log Action dialog is displayed.



This dialog provides default values for the three standard event log parameters—Source, Type, and Event—and allows you to change them.

3. Leave the default value in the **Source** text field, or type in a new registered source.
4. Select one of the standard event log types from the **Type** drop-down list.
Select the most appropriate option for the situation your alarm transition detects. The options are Error, Warning, Informational, Audit Success, and Audit Failed.
5. Leave the default event ID in the **Event** field, or type a new one.
6. Select the **OK** button in the Event Log Action dialog.
7. Select the **OK** button in the Transition Definition window.
8. Select the **Save** button in the Alarm Definition window.

Fire Trigger

In NerveCenter, you have several ways of generating a trigger. For instance, you can use a poll, a mask, or the `FireTrigger()` function to fire the trigger. You can also use the Fire Trigger alarm action to produce a trigger. This action is useful when you need one alarm to send a trigger to itself or to another alarm.

Here are some examples of when you might need to use the Fire Trigger alarm action:

- You want an alarm transition to fire a trigger on a delayed basis so that your alarm will know when a certain amount of time has passed.

This strategy is used in the predefined alarm `IfLinkUpDown`, shown in [Figure 47](#).

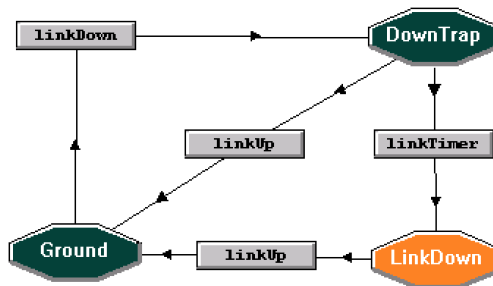


Figure 47: IfLinkUpDown Alarm

On the `linkDown` transition, this alarm fires the `linkTimer` trigger on a three-minute delay. If a `linkUp` trigger does not cause a transition to `Ground` within three minutes, the `linkTimer` trigger is fired, and the alarm transitions to the `LinkDown` state.

Using the action for its timing capabilities is the most common use of the Fire Trigger action.

- You want to send information to an alarm instance about an event that is outside its scope. As an example, let's look at the predefined alarm `BetterNode`, which tracks the status of a node on a different subnet from the NerveCenter server.

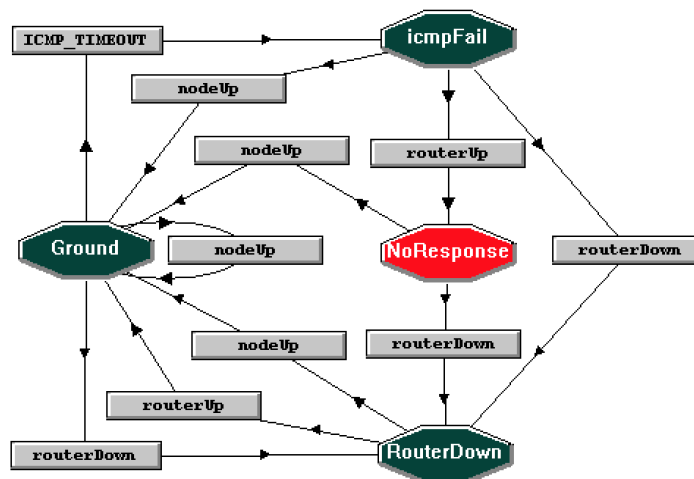


Figure 48: BetterNode Alarm

If NerveCenter is unable to ping a node, the node's alarm instance transitions to the `lcmpFail` state. What happens next, however, depends on a trigger fired by an alarm instance monitoring the router that sits between the NerveCenter server and the node being monitored with BetterNode. If the alarm instance monitoring the router generates the `routerUp` trigger, the BetterNode alarm transitions to the critical `NoResponse` state, but if the router's alarm generates the `routerDown` trigger, the BetterNode alarm transitions to the normal `RouterDown` state.

- A behavior model requires alarms of different scopes to detect a condition.

For example, suppose you want to create a behavior model that detects high interface traffic at the node level. You'll need to create a subobject scope alarm that detects high traffic on a single interface and fires a trigger that notifies a node scope alarm that the interface is busy. You'll also need a node scope alarm that tracks the triggers being fired by the subobject scope alarms. Behavior models of this type are called *multi-alarm behavior models*.

Note: If you later turn an alarm off or reset the alarm to ground, any pending triggers fired by that alarm are cleared if the **Clear Triggers for Reset To Ground or Off** checkbox is checked in the alarm's definition window.

TO ADD A FIRE TRIGGER ACTION TO AN ALARM TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Fire Trigger** from the pop-up menu.
The Fire Trigger Action dialog is displayed.

Fire Trigger Action

Trigger Name

SubObject

Node

Property

Delay

Days Hours Seconds

OK Cancel Help

3. In the **Trigger Name** field, specify the name of the trigger to be fired when the transition occurs. You can either type in the name of a new or existing trigger or select the name of an existing trigger from the Trigger Name drop-down list.
4. Either leave the default values in the **SubObject**, **Node**, and **Property** fields, or enter new values using the keyboard or the associated drop-down lists.

If you want your Fire Trigger action to simply provide a timer for its own alarm instance, the default values are fine. The defaults ensure that the resulting trigger affects only alarm instances concerning the same node and subobject as the current alarm instance.

If the trigger being fired will affect instances of a different alarm, you may need to change the default values. The steps below explain the values you can provide for these attributes.

- a. To change the value in the **SubObject** field, either type in a new value or select a value from the **SubObject** drop-down list.

Note: When choosing a SubObject value, keep in mind that alarm instances with subobject scope must reference the same subobject to be transitioned by this trigger. For transitions with instance scope, only the instances must match; the base objects can be different. Any alarm instances with a node or enterprise scope will ignore the value in the **SubObject** field.

The following table lists the acceptable values for the SubObject field.

Value	Explanation
\$SO	The trigger inherits the originating alarm's subobject. This is the default.
\$ANY	The trigger is assigned a subobject that matches any destination alarm subobject. Think of this as a subobject wildcard.
\$ON.\$OI	If the originating alarm has a subobject that consists of a base object plus an instance joined by a period, the trigger inherits the originating alarm's subobject (same as \$SO). However, if the originating alarm does not have this type of subobject, the trigger's subobject is null (see \$NULL below).
\$ON	If the originating alarm has a subobject that consists of a base object plus an instance joined by a period, the trigger inherits the base object portion of the alarm's subobject and appends to this base object a period and a wildcard for the instance. The resulting trigger can drive alarm instances with a subobject containing a matching base object and any instance. For example, let's say that an alarm instance with the subobject ifEntry.3 fires a trigger using \$ON. The trigger's subobject will be ifEntry.*, and the trigger will affect alarm instances with subobjects such as ifEntry.1, ifEntry.2, and so on. If the originating alarm instance does not have a subobject that consists of a base object plus an instance, \$ON is equivalent to \$NULL.
\$NULL	The trigger is assigned a null subobject. The only subobject scope alarm that can be affected by such a trigger is one that has a null subobject itself.
<i>baseObject.instance</i>	You can type the subobject. The trigger's subobject is set to the subobject you specify, for example, ifEntry.3 or system.0.

Value	Explanation
<i>anyString</i>	This feature enables you to take advantage of the matching rules for triggers and alarm transitions by making creative use of the subobject attributes of these objects. For example, you could use the name of an application as the subobject in order to correlate all events relating to that application.

- b. To change the value of the **Node** field, either type in a new value or select a value from the **Node** drop-down list.

The following table lists the acceptable values for the Node field.

Value	Explanation
\$NODE	The trigger inherits the originating alarm instance's node. This is the default.
\$ANY	The trigger is assigned a node that matches any destination alarm instance node. Think of this as a node wildcard.
<i>nodeName</i>	You assign the name of any managed node to this attribute. Use the Node drop-down list to prevent spelling errors.

- c. To change the value of the **Property** field, either type in a new value or select a value from the **Property** drop-down list.

The following table lists the acceptable values for the Property field.

Value	Explanation
\$PROPERTY	The trigger inherits the originating alarm instance's property. This is the default.
\$NO_PROP	The trigger is assigned no property. In this case, NerveCenter ignores the trigger's property attribute when determining which alarm transitions the trigger can affect.
<i>property</i>	The trigger is assigned the property you type in or select from the Property drop-down list.

When a trigger contains a property, the property group of the node found in a destination alarm instance's node data member must contain the trigger's property. Otherwise, no alarm transition will occur.

5. Select a delay for the trigger by entering a positive integer in the **Delay** text field and selecting the appropriate radio button: **Days**, **Hours**, **Minutes**, or **Seconds**.
6. Select the **OK** button in the Fire Trigger Action dialog.
7. Select the **OK** button in the Transition Definition window.
8. Select the **Save** button in the Alarm Definition window.

Forward Trap

When you define a Forward Trap action, you cause NerveCenter to set up for reproducing and sending a received SNMP Trap or Inform onward to the destinations that have been defined by the NerveCenter Administrator application. The Forward Trap action applies only to contexts where an alarm transition is being made in direct response to an SNMP Trap or Inform having been received. The Forward Trap action has no effect if it is encountered on an alarm transition where the stimulus was something other than a SNMP Trap or Inform.

Note: Note: Traps and Informs are forwarded to a set of destinations that have been prepared using the NerveCenter Administrator. Using the Administrator program a table of Trap Destinations is prepared. Each destination is defined with a name.

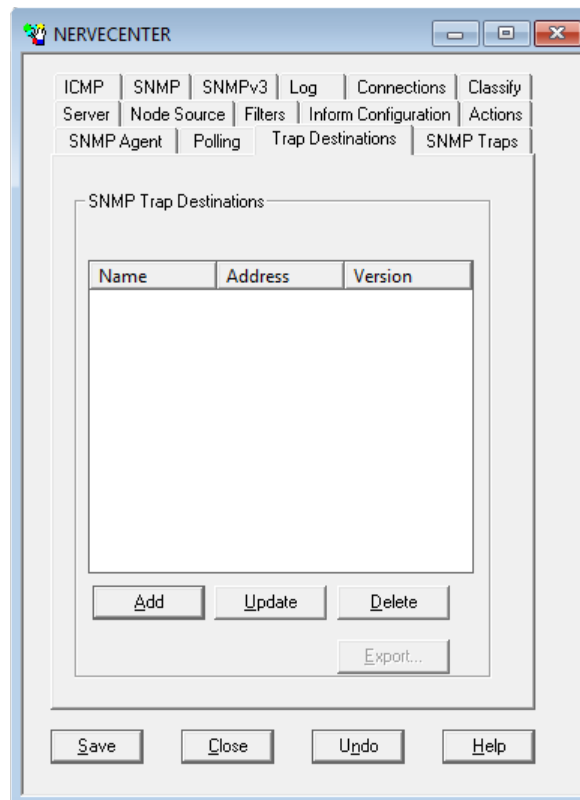


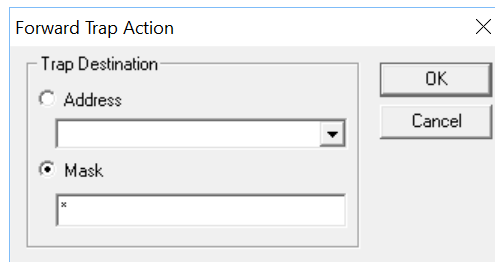
Figure 49: Administrator with three defined trap destinations

When a Forward Trap operation occurs in response to a received SNMP Trap or Inform, the configured Forward Trap action causes NerveCenter to match the naming provided by the action against the names provided in the Trap Destinations table — each match will cause NerveCenter to forward the Trap or Inform to that destination.

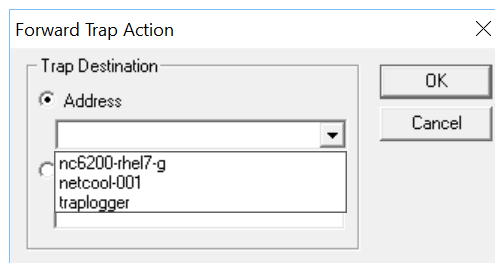
The Forward Trap Action can be set up to match the set of names from the Trap Destinations table either exactly or using a wildcard.

TO ADD A FORWARD TRAP ACTION TO AN ALARM TRANSITION

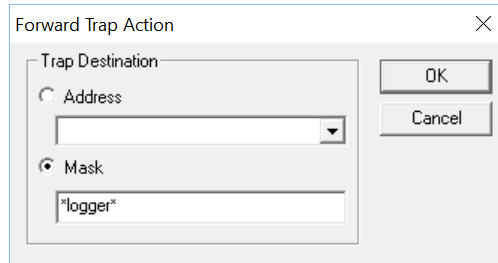
1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Forward Trap** from the pop-up menu.
The Forward Trap Action dialog is displayed.



3. Configure the Trap Destination.
 - For an exact match:
 - a. Select the **Address** radio button and then select an option from the list to match against.
The list displays Trap Destination names as defined in the Administrator.



- b. Select the **OK** button to close the dialog and add the Forward Trap action to the set of actions to be performed on this transition.
When this action occurs at runtime, the received Trap or Inform will be relayed forward to this (and only this) named destination.
 - To use wildcard matching:
 - a. Select the **Mask** radio button and enter the wildcard specification in the text field.
The mask uses the typical shell prompt format: ? matches a single character and * matches zero or more characters. The default is * which will match against all names in the Trap Destination table. For example, a value such as *logger* would lead to a runtime match of all Trap Destination table entries that have “logger” as part of their name.



- b. Select the **OK** button to close the dialog and add the Forward Trap action to the set of actions to the set of actions to be performed on this transition.
 4. Select the **OK** button in the Transition Definition window.
 5. Select the **Save** button in the Alarm Definition window.
-

Inform

Once NerveCenter has used its event-correlation abilities to detect a problem, it can notify a network management platform or another NerveCenter server of the problem using the Inform action. This alarm action enables you to notify another NerveCenter when a significant network event is detected.

Note: For information about sending messages to IBM Tivoli Netcool/OMNIbus, see the section [Inform Platform on page 289](#).

Inform sends the equivalent of an SNMP trap to its recipients, and the specific trap number in the trap indicates the nature of the problem. The recipients of the trap must be set up to interpret this trap properly and to take appropriate action.

Note: Although the message that the Inform action sends to its recipients contains the same information as a trap, the message is not sent via UDP. Because the delivery mechanism must be reliable, the message is sent via TCP.

Typically, a behavior model uses the Inform alarm action on a transition to some terminal state. For example, consider the predefined alarm SnmpStatus, shown in Figure 50.

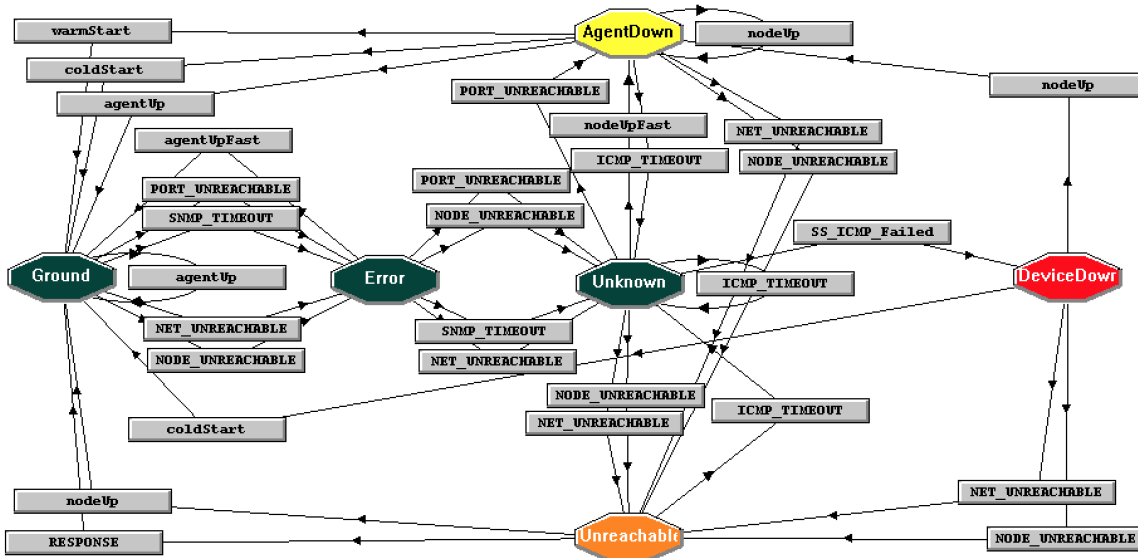


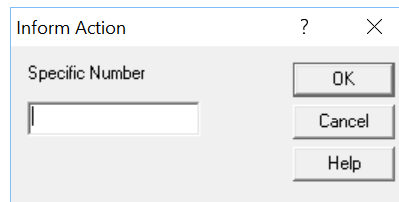
Figure 50: SnmpStatus Alarm

Only one transition in this alarm contains an Inform action. That is the transition `SS_ICMP_Failed`, which leads to the `DeviceDown` state. An alarm does not specify who is to receive Inform messages. The recipients of these messages are set up in the NerveCenter Administrator by the person who configures NerveCenter.

If the destination is a network management platform, you must create a new event message for the platform that will be posted when it receives your Inform message. If the destination is another NerveCenter server, you must create a trap mask in the destination NerveCenter to capture the Inform message. (For information on how to create such a trap mask, see the section [Creating a Trap Mask](#) on page 230.)

TO ADD AN INFORM ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Inform** from the pop-up menu.
The Inform Action dialog is displayed.



3. Type a number in the range 100000 to 199999 in the **Specific Number** text field, or leave this field blank.
For more information about the specific number field, see [Inform Specific Numbers on page 290](#).
4. Select the **OK** button in the Inform Action dialog.
5. Select the **OK** button in the Transition Definition window.
6. Select the **Save** button in the Alarm Definition window.

As mentioned earlier, each Inform message looks like an SNMP trap. Thus, it contains a great deal of information in addition to a specific-trap number. You can use this information in a NerveCenter trap mask. This information is listed below:

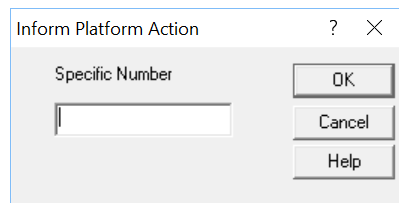
- A timestamp.
- A generic trap number. This number will always be 6.
- An enterprise. The enterprise OID will always be 1.3.6.1.4.1.78.
- A list of variable bindings. For a list of these variable bindings, see the section [Variable Bindings for NerveCenter Informs on page 234](#).

Inform Platform

You can design alarms to notify the IBM Tivoli Netcool/OMNIBus network management platform of significant events that require your attention. In addition, you must have a corresponding network management platform event configured to listen for the specific trap number.

TO CONFIGURE AN INFORM PLATFORM ACTION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Inform Platform** from the pop-up menu.
The Inform Platform Action dialog is displayed.



3. In the **Specific Number** field, enter the specific trap value you want to use for the Inform. Typically, this number would be between 100000 and 199999.
For more about the specific number field, see [Inform Specific Numbers on the next page](#).
4. Select the **OK** button.
The new action is added to the transition.
5. Select the **OK** button again to close the Transition Definition dialog box and save your action.

Note: When you are finished making changes to an alarm's definition, select Save to save all changes before closing the Alarm Definition window.

Inform contains the following information in addition to the specific trap number you enter:

- A timestamp.
- A generic trap number. This number will always be 6.
- An enterprise. The enterprise OID will always be 1.3.6.1.4.1.78.
- A list of variable bindings. For a list of these variable bindings, see the section [Variable Bindings for NerveCenter Informs on page 234](#).

The following information about the MIB objects is inserted at the end of the variable bindings.

```
ObjectName1.ObjectInstance1=Value1;ObjectName2.ObjectInstance2=Value2; ...
```

A NerveCenter MIB definition for these variable bindings is located in `/opt/OSInc/mibs/nervectr.mib`

If you incorporate this MIB into your network management platform, the attribute names of the variable bindings are displayed in the platform's message browser.

Inform Specific Numbers

When creating an Inform or Inform Platform action, you are expected to supply a Specific Number for the Inform. Normally, you should enter a number in the range 100000 to 199999 or leave this field blank. The trap numbers 0 to 99999 are reserved for NerveCenter use, and the numbers 200000 and above are reserved for future use.

The number you supply becomes the specific trap number in the trap-like message that is sent to all the destinations that have been configured to receive Inform messages. If the destination is a network management platform, you must create a new event message for the platform that will be posted when it receives your Inform message.

If the destination is another NerveCenter server, you must create a trap mask in the destination NerveCenter to capture the Inform message. (For information on how to create such a trap mask, see the section [Creating a Trap Mask on page 230](#).) The portion of NerveCenter that must be installed with a network management platform defines general event messages for these default specific-trap values. However, other NerveCenter servers know nothing of default values in Inform messages sent by this NerveCenter server. For that reason, you must create a trap mask in the destination NerveCenter to receive the Inform message.

If you leave the Specific Number field blank, NerveCenter supplies a default specific trap number. NerveCenter creates this default value by adding 1000 to the severity level of the destination alarm state. Thus, if the Inform action takes place on a transition to a Critical state, the default specific number is 1012, because the severity level of Critical is 12.

You can determine a severity's number by choosing **Severity List** from the client's **Admin** menu.

When NerveCenter sends Informs to your platform, NerveCenter first checks the minimum severity value configured in NerveCenter Administrator to ensure that the trap value for the Inform matches or exceeds that severity. There is one case when NerveCenter disregards the minimum severity value specified in Administrator: After NerveCenter sends an Inform, if the condition returns to a normal state—that is, a state below the minimum severity threshold you configure—it's important that NerveCenter notify the platform of this change. Therefore, if a node transitions the alarm from a severity above the minimum value to a severity below the minimum value, and the transition includes an Inform action, NerveCenter will send a Normal Inform to the platform. This allows the platform to reset the mapped severity color associated with the node.

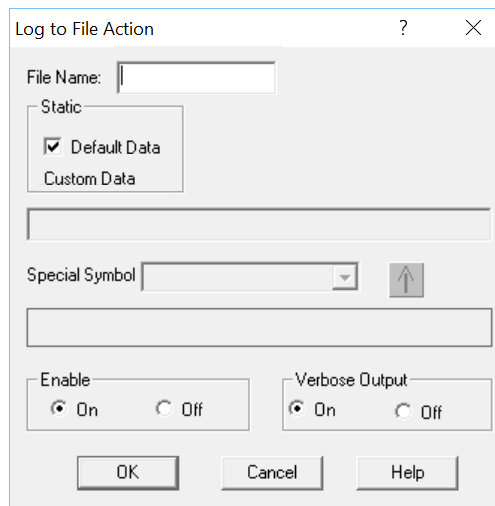
Log to File

The Log to File alarm action writes information about an alarm transition to an ASCII text file.

Note: Over use of Log to File may slow down NerveCenter's performance.

TO ADD A LOG TO FILE ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Log to File** from the pop-up menu.
The Log to File Action dialog is displayed.



3. In the **File Name** text field, type in either a filename or a full pathname for your log file.
If you enter a filename, the log file is written to *install_directory/opt/OSInc/userfiles/logs*. If you enter a full pathname, the log file is written to the directory you specify.
4. To log particular information instead of NerveCenter's default data, do the following:
 - a. Deselect the **Default Data** checkbox.
 - b. In the **Custom Data** field, type or paste the variables you want included in the log, separating each variable with a space.

[Table 23](#) lists the variables you can include in the Custom Data field.

Note: You can also select a variable from the Special Symbol drop-down listbox and then click the red arrow.

5. Select either the **On** or **Off** radio button in the Enable frame.
This option gives you the ability to disable logging without disabling the alarm of which the logging action is a part.
 6. Select either the **On** or **Off** radio button in the Verbose Output frame.
If you turn Verbose Output on, NerveCenter labels each value it writes to the log file. Otherwise, NerveCenter writes only the values, separated by commas, to the log file. This may be what you want if you are using the log file only as the basis for reports.
 7. Select the **OK** button in the Log to File Action dialog.
 8. Select the **OK** button in the Transition Definition window.
 9. Select the **Save** button in the Alarm Definition window.
-

See "[NerveCenter Variables](#)" on page 301 for a list of the values that constitute a log file entry. And remember that if you create your log file in non-verbose mode, the values in an entry are not labeled; they are separated by commas. You may need to refer to the table mentioned above to interpret the contents of a log entry. [Table 23](#) also lists the variables you can use in log actions that log alarm data to a file or database.

Notes

Whenever you create an alarm, you can—and should—create *notes* that document the alarm. Generally, this documentation should accomplish the following goals:

- Explain the purpose of the alarm
- Briefly describe the alarm's states and transitions
- List the polls, masks, and alarms that fire triggers that can affect the alarm
- Describe the actions that take place during alarm transitions, especially Fire Trigger and Perl Subroutine actions
- Document any programs or scripts that are called via a Command action
- Name any reports that are run against data logged by the alarm
- Explain any customization required to work with the alarm

Using the Notes alarm action you can cause an alarm's notes to be displayed by a behavior model. The notes are displayed in the Alarm Definition Notes window when the transition with which the Notes action is associated occurs. For example, adding a Notes action to the first transition in the predefined alarm IfDataLogger would cause the notes in Figure 51 to be displayed whenever that transition occurred:

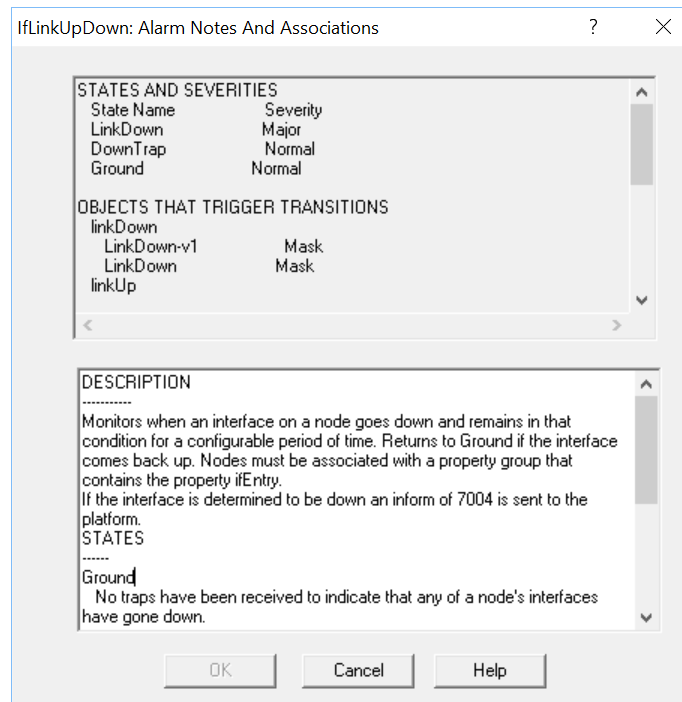


Figure 51: Notes for IfData_LogToFile Alarm

TO ADD THE NOTES ALARM ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Notes** from the pop-up menu.
The Notes action is added to the list of actions in the Transition Definition window.
3. Select the **OK** button in the Transition Definition window.
4. Select the **Save** button in the Alarm Definition window.

Perl Subroutine

The Perl Subroutine alarm action enables you to execute a Perl subroutine when a particular alarm transition occurs. This action is similar to the Command action in that it enables you to execute a script. However, the Perl Subroutine action can be much more powerful than the Command action because:

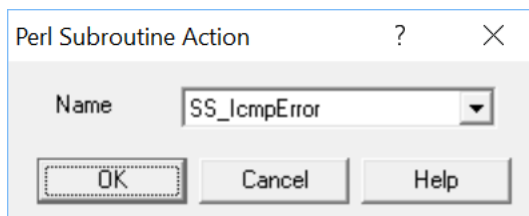
- NerveCenter provides a set of functions for use in Perl subroutines. These functions enable you to access the contents of a trigger's variable bindings, fire triggers, assign property groups to nodes, and so on. For complete information about these functions, see the section [Functions for Use in Perl Subroutines](#) on page 297.
- NerveCenter provides a set of variables for use in Perl subroutines that give you access to a great deal of internal NerveCenter information about the alarm transition that just occurred. For details, see the section [NerveCenter Variables](#) on page 301.

Note: You can call Perl subroutines defined outside of NerveCenter from the command line; however, these Perl subroutines use the Perl interpreter installed by the user and not the Perl engine embedded in NerveCenter. Also, Perl programs run outside of NerveCenter will not have access to any NerveCenter variables or data structures.

Using these functions and variables, you can create scripts that you could not write using another language. The section [Perl Subroutine Example](#) on page 303 presents an example of how you might use the Perl Subroutine action.

TO ADD A PERL SUBROUTINE TO A TRANSITION

1. Define the Perl subroutine. This task is documented in the section [Defining a Perl Subroutine](#) on the facing page.
2. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
3. Select **Perl Subroutine** from the pop-up menu.
The Perl Subroutine Action dialog is displayed.



4. Select a Perl subroutine from the **Name** drop-down list.
This list contains all the compiled Perl subroutines stored in the NerveCenter database.
5. Select the **OK** button in the Perl Subroutine Action dialog.
6. Select the **OK** button in the Transition Definition window.
7. Select the **Save** button in the Alarm Definition window.

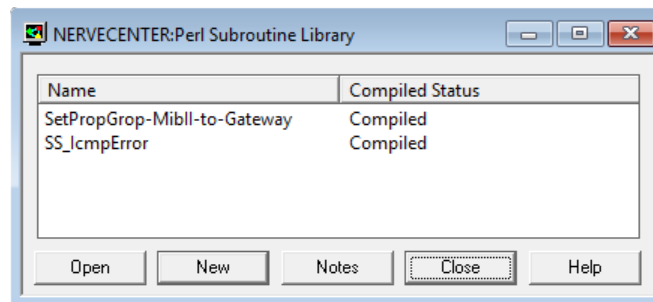
Defining a Perl Subroutine

Before you can add a Perl subroutine to a transition, you must write it (obviously) and store a compiled version of it in the NerveCenter database.

Note: Perl subroutines that you define inside NerveCenter use the Perl engine embedded in NerveCenter and *not* any Perl interpreters installed outside of NerveCenter. Any Perl programs run outside of NerveCenter will not have access to any NerveCenter variables or data structures.

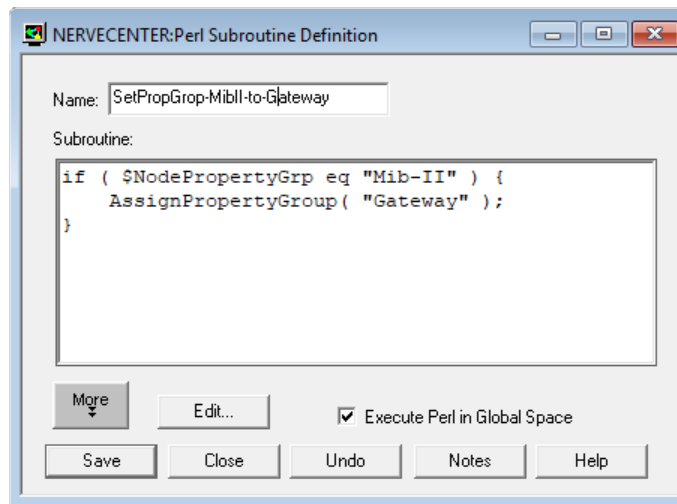
TO DEFINE A PERL SUBROUTINE WITHIN NERVECENTER

1. From the **Admin** menu in the main client window, choose **Perl Subroutine List**.
The Perl Subroutine List window appears.



This window contains a list with an entry for each Perl subroutine defined in your NerveCenter database. The Compiled Status column indicates whether the subroutine has been successfully compiled. From this window, you can add a new subroutine, modify an existing subroutine, or view the notes for a subroutine.

- To add a new subroutine to NerveCenter, select the **New** button.
The Perl Subroutine Definition window appears.



This window enables you to name and define a new Perl subroutine.

- Type the name of your new Perl subroutine in the **Name** field.

Note: The maximum length for Perl subroutine names is 255 characters.

- To document your Perl subroutine, select the **Notes** button, enter a description in the Perl Subroutine Notes window, and select the **OK** button in that window.
- Type your Perl subroutine in the **Subroutine** text entry box.

Use Perl version 5 to write your subroutine. You can also make use of the NerveCenter functions and variables discussed in the sections [Functions for Use in Perl Subroutines on the facing page](#) and [NerveCenter Variables on page 301](#)

If you right-click in the Perl-subroutine editing area, you'll see a pop-up menu that lists all the functions and variables available for writing Perl subroutines. For more information about this pop-up menu, see the section [Using the Pop-Up Menu for Perl on page 182](#).

Note: The maximum length for identifiers in Perl subroutines is 251 characters (252 including the variable type identifier character \$, %, and so on).

- If you want to use the shared Perl interpreter, select the **Execute Perl in Global Space** checkbox.

Note: If you select Execute Perl in Global Space, the poll condition executes in a shared Perl interpreter. You can use Global variables in your poll condition to share information between other Perl routines such as trigger functions or Perl subroutines, however, Perl intensive poll conditions may impede NerveCenter's performance.

If you do not select Execute Perl in Global Space, the poll condition executes in a Perl interpreter dedicated to poll conditions. This will improve NerveCenter's performance, however you cannot use global variables in your poll condition to share information between other Perl routines such as trigger functions or Perl subroutines.

For more information about the various Perl interpreters, see [NerveCenter and Perl on page 45](#).

7. Select the **Save** button in the Perl Subroutine Definition window.

NerveCenter automatically attempts to compile the subroutine. If your Perl subroutine does not compile correctly, NerveCenter displays an error message from the Perl compiler. It also saves the subroutine and places it in the Perl Subroutine List, with the Compiled Status listed as Not Compiled.

If your Perl subroutine compiles successfully, the saved subroutine is available for use in a Perl Subroutine alarm action. It won't be executed unless it's made the object of a Perl Subroutine action and the associated alarm transition occurs.

Caution: Do not call the `exec` or `exit` function from within your Perl subroutine. These statements may cause the NerveCenter server to terminate.

Functions for Use in Perl Subroutines

NerveCenter provides a number of functions that you can use in your Perl subroutines. The list below indicates what types of functions are available and where you can find detailed information about each function:

- **Variable-binding functions.** These functions enable you to determine the number of variable bindings in a trigger's variable-binding list and to obtain information about each variable binding. For instance, you can retrieve the subobject and attribute associated with a variable-binding and the value of a variable-binding.

For reference information about these functions, see the section [Variable-Binding Functions on page 204](#).

- **String-matching functions.** These functions enable you to determine whether a string contains another string or a particular word. The functions are useful in conditions that test the value of a variable binding for a substring.

For reference information about these functions, see the section [String-Matching Functions on page 181](#).

- **DefineTrigger()**. This function enables you to create triggers which you can assign to variables and fire using `FireTrigger()` in NerveCenter Perl expressions.
For reference information about this function, see the section [DefineTrigger\(\) Function on page 175](#).
- **FireTrigger()**. This function enables you to fire a trigger from your Perl subroutine. You can specify the name, subobject, and node attributes of the trigger.
For reference information about this function, see the section [FireTrigger\(\) Function on page 177](#).
- **AssignPropertyGroup()**. This function enables you to assign a property group to the node associated with a trigger.
For reference information about this function, see the section [AssignPropertyGroup\(\) Function on page 175](#).
- **in()**. This function determines whether one scalar value is in a set of scalar values.
For reference information about this function, see the section [in\(\) Function on page 179](#).
- **AddNode()**. This function enables you to add a node to the NerveCenter managed node list from a NerveCenter Perl expression.
For reference information about this function, see the section [AddNode\(\) Function on the facing page](#).
- **Counter()**. This function returns the current value of an alarm counter. For reference information about this function, see the section [Counter\(\) Function on the facing page](#).
- **NC::AlarmCounters**. This function enables you to do any of the following: increment alarm counters by a number other than one, decrement alarm counters by a number other than one, create alarm counters, set alarm counters, and retrieve alarm counters. For reference information about this function, see the section [NC::AlarmCounters on page 179](#).
NC::AlarmCounters are independent of and not related to the alarm action Alarm Counter. See [Alarm Counter on page 271](#) for details.
- **Node relationship functions**. These functions enable you to import, export, and delete node parenting relationships from the NerveCenter database. You can use these functions in Perl subroutines that are called from alarms that you transition on demand. [Node Relationship Functions on page 300](#).

AddNode() Function

The AddNode() function adds a node to the NerveCenter managed node list. This function can be called from a poll condition, trap mask trigger function, or a Perl Subroutine alarm action.

Syntax: AddNode("*node name*");

Arguments: node name is the IP address of the node to add to the managed node list and must be a valid IP address enclosed in quotes, for example, "123.123.123.123".

Description: Adds a node to the NerveCenter managed node list, with the following attributes:

- Node name must be a valid IP address enclosed in quotes (for example "192.168.1.1").
- The address assigned to the node will be the address specified for the node name.
- The node property group will be assigned to the "NCDefaultGroup"
- The node community string will be assigned the default V1 community string value ("public").
- The node will be marked as 'managed' and 'not suppressed'
- The SNMP version of the node will be assigned as V1.
- No action is taken if a node of the same name already exists in the node list. No validation that the node name doesn't already exist is performed at compile time.

Counter() Function

You use the Counter() function to get the value of an alarm counter for a particular alarm instance. The function can only be called from a Perl Subroutine alarm action or an Action Router rule.

The syntax of the Counter() function is shown below:

Counter()

Syntax: Counter("*counterName*")

Arguments:

counterName - The name of an existing alarm counter.

Description: The function returns the value of the specified counter.

Node Relationship Functions

The following functions import, export, and delete node parenting relationships from the NerveCenter database. You can use these functions in Perl subroutines that are called from alarms that you transition on-demand. One use for these functions is with the downstream alarm suppression behavior model that is shipped with NerveCenter. For more information, refer to "[Downstream Alarm Suppression](#)" on page 383.

LoadParentsFromFile()

Syntax: LoadParentsFromFile(*filename*)

Arguments:

filename - The name of the OVPA or manually created file containing the child parent relationships. This file should list each child node followed by the parent nodes in space-delimited fashion.

Description: Imports an OVPA or manually created file containing node parenting relationship information into the NerveCenter database.

Example: This statement loads the node relationship file data from the file nodeparents.dat into the NerveCenter database:

```
NC:: LoadParentsFromFile (nodeparents.dat)
```

DumpParentsToFile()

Syntax: DumpParentsToFile(*filename*)

Arguments:

filename - The name of the file NerveCenter will output containing the child parent relationships exported from NerveCenter database.

Description: Exports node parenting relationship information from the NerveCenter database to the specified file on the local machine.

Example: This statement exports node relationship information from the NerveCenter database to the file nodeparents.dat:

```
NC:: DumpParentsToFile (nodeparents.dat)
```

RemoveAllParents()

Syntax: RemoveAllParents()

Description: Deletes node parenting relationship information from the NerveCenter database.

Example: This statement deletes node relationship information from the NerveCenter database.

```
NC:: RemoveAllParents
```


NerveCenter Variables

NerveCenter defines a number of variables for use in Perl subroutines, Command Alarm actions, and logging actions. These variables contain information about the alarm transition that just occurred and about the trigger that caused the transition.

The variables (and functions) available to you for use in poll conditions, trigger functions, Action Router rule conditions, and Perl Subroutine alarm actions are summarized in a pop-up menu for Perl accessible via a right mouse click in the respective editing area. (See the section, [Using the Pop-Up Menu for Perl on page 182](#), for more information.)

The variables available to you for use in Command Alarm actions and the logging actions are available to you via the Special Symbol drop-down listbox.

The complete list of NerveCenter variables that you can use are shown in [Table 23](#):

Table 23: NerveCenter Variables

Variable	Contains
<code>\$AlarmInstanceID</code>	String. The unique identifier for an alarm instance managed by a NerveCenter Server. If you are connected to more than one server, you can use the <code>\$NCHostName</code> variable to identify the server associated with the alarm instance.
<code>\$AlarmName</code>	String. The name of the alarm whose instance just underwent a transition.
<code>\$AlarmProperty</code>	String. The name of the alarm's property.
<code>\$AlarmTransitionTime</code>	String. The time at which the alarm transition occurred. This time is formatted as follows: <i>mm/dd/yyyy hh:mm:ss day</i> . An example of an alarm transition time is 03/31/2017 11:02:26 Fri.
<code>\$Date</code>	Number. The date on which the alarm transition occurred. When you use this variable in a comparison, compare it to a value of the form <i>mm/dd/yyyy</i> . Before using this value in the comparison, NerveCenter converts it to a number of seconds since January 1, 1970.
<code>\$DayOfWeek</code>	Number. The day of the week on which the alarm transition occurred. When you use this variable in a comparison, compare it to one of the following values: SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY. These values are converted to numbers between 0 and 6 before they are used in the comparison.
<code>\$DestState</code>	String. The state of the alarm instance following the current transition.
<code>\$DestStatePlatformSev</code>	String. The network management platform severity that is mapped to <code>\$DestStateSev</code> .
<code>\$DestStateSev</code>	String. The severity of the state where the transition ended.
<code>\$NCHostName</code>	String. The NerveCenter Server associated with an alarm instance.

Variable	Contains
\$NewMaxNodePlatformSev	String. The network management platform severity that is mapped to \$NewMaxNodeSev.
\$NewMaxNodeSev	String. The maximum severity associated with a node, following the current transition. This maximum severity is determined by looking at the states of all alarm instances that are monitoring the node.
\$NodeAddress	String. The IP address of the node being monitored.
\$NodeAddressList	String. A comma-separated list of all the IP addresses associated with the node being monitored. No white space is allowed in this list.
\$NodeName	String. The name of the node being monitored by the alarm instance that underwent the transition.
\$NodePropertyGrp	String. The property group of the node being monitored.
\$NoOfVarBinds	Number. The number of variable bindings in the trigger that caused the alarm transition. These variable bindings may have been derived from a poll condition or an SNMP trap.
\$OrigState	String. The state of the alarm instance prior to the current transition.
\$OrigStatePlatformSev	String. The network management platform severity that is mapped to \$OrigStateSev.
\$OrigStateSev	String. The severity of the state where the transition began.
\$PollKey	String. If a poll caused the transition, this variable contains a value that uniquely describes the poll and the alarm instance with which it interacted. That value has the format <i>pollID.nodeID.baseObject.instance</i> . \$PollKey is usually used as an index into a Perl hash.
\$PrevMaxNodePlatformSev	String. The network management platform severity that is mapped to \$PrevMaxNodeSev.
\$PrevMaxNodeSev	String. The maximum severity associated with a node, prior to the current transition. This maximum severity is determined by looking at the states of all alarm instances that are monitoring the node.
\$ReadCommunity	String. The read community string of the node being monitored.
\$Time	Number. The time at which the alarm transition occurred. When you use this variable in a comparison, compare it to a value of the form <i>hh:mm</i> . NerveCenter converts this value to a number of seconds before performing the comparison.
\$TrapPduAgentAddress	String. If an SNMP trap caused the transition, this variable contains the trap's agent address.
\$TrapPduCommunity	String. If an SNMP trap caused the transition, this variable contains the trap's community string.

Variable	Contains
\$TrapPduEnterprise	String. If an SNMP trap caused the transition, this variable contains the trap's enterprise OID.
\$TrapPduGenericNumber	Number. If an SNMP trap caused the transition, this variable contains the trap's generic trap number.
\$TrapPduSpecificNumber	Number. If an SNMP trap caused the transition, this variable contains the trap's specific trap number.
\$TrapPduTime	Number. If an SNMP trap caused the transition, this variable contains the trap's timestamp.
\$TriggerBaseObject	String. The base object portion of the trigger's subobject attribute. For example, if the trigger's subobject is IfEntry.2, the base object is ifEntry.
\$TriggerInstance	Number. The instance portion of the trigger's subobject attribute. For example, if the trigger's subobject is IfEntry.2, the instance is 2.
\$TriggerName	String. The name of the trigger that caused the alarm transition.
\$VarBinds	String. The list of all variable bindings in the form attribute=value. In the case of Perl subroutines and Action Router rules, it makes sense to use attribute name, value or object for an individual variable binding.
\$VB(n)	String. The nth variable binding. You can use \$VB(n) in Log to File and Log Database alarm actions only.
\$WriteCommunity	String. The write community string of the node being monitored.

Perl Subroutine Example

As a simple example, suppose that you want to poll a node for the value of an attribute and to fire different triggers depending on the value. Let's say that you're interested in the value of `ifEntry.ifOperStatus` and that you want to fire different triggers for the values 1 (up), 2 (down), and 3 (testing). You also want to fire a fourth trigger if the value is some other number.

You could solve this problem by using multiple polls with the poll conditions `ifEntry.ifOperStatus == 1`, `ifEntry.ifOperStatus == 2`, and so on. However, this would be very inefficient. A better solution would be to use the poll to retrieve the value of the attribute and to fire a trigger if it is successful. So the poll condition would simple be:

```
ifEntry.ifOperStatus present
```

Then, on the transition associated with the poll's trigger, you could execute a Perl subroutine. This subroutine might look something like this:

```
if (ifEntry.ifOperStatus == 1) {
    FireTrigger("OperStatusUp");
}
elsif (ifEntry.ifOperStatus == 2) {
    FireTrigger("OperStatusDown");
}
elsif (ifEntry.ifOperStatus == 3) {
    FireTrigger("OperStatusTest");
}
else {
    FireTrigger("OperStatusBad");
}
```

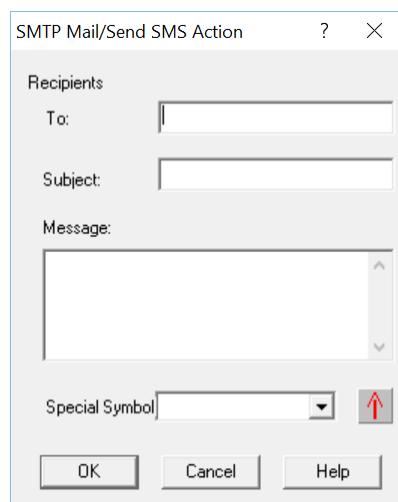
Send SMS

The Send SMS alarm action enables an alarm to send an SMS message concerning a transition.

Note: Before you can use the Send SMS action, NerveCenter must specify an SMTP server. This setup is covered in [Specifying an SMTP Server for Mail Notification in Managing NerveCenter](#).

TO ADD A SEND SMS ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Send SMS** from the selection control.
The SMTP Mail/Send SMS Action dialog is displayed.



The screenshot shows a dialog box titled "SMTP Mail/Send SMS Action". It features a "Recipients" section with a "To:" text field and a "Subject:" text field. Below these is a "Message:" section with a large text area. At the bottom, there is a "Special Symbol" dropdown menu with an upward arrow icon, and three buttons: "OK", "Cancel", and "Help".

3. In the **To:** field, enter the SMTP gateway addressing for reaching your SMS target per the instructions of your carrier service.

Optionally, fill the **Subject:** field. Any entry here will be included at the start of the resulting SMS message. As SMS messages need to be short (140 characters), this field can be skipped and left blank.

4. Enter the message body in the **Message:** field.

The message text (limited to 140 characters, including the Subject you entered one) may be used as boilerplate wherein you insert a supported variable in the text, which is replaced at runtime with values taken from the live context.

To add a variable, position the cursor in the text and select a variable from the **Special Symbol** list. Then click the 'up' arrow to insert the selected variable into the text.

5. Select the **OK** button.

The new action appears in the list of actions in the Transition Definition window.

6. Select the **OK** button in the Transition Definition window.

7. Select the **Save** button in the Alarm Definition window.

Send Trap

The Send Trap alarm action enables you to send an SNMPv1 trap when a transition occurs and gives you virtually complete control over the contents of the trap.

Note: NerveCenter does not send SNMPv3 traps, because under SNMPv3, a node's IP address is no longer sent in the packet's header; therefore, NerveCenter cannot simulate a node's IP address and send the SNMPv3 trap.

Generally, when one alarm must communicate with another, the first uses the Fire Trigger action to fire a trigger that causes a transition in the second. However, Send Trap can also be used for this type of inter-alarm communication. The first alarm can send a trap to the NerveCenter server, the server can process the trap using a trap mask (which can fire a trigger), and the trigger can cause a transition in the second alarm. This is a more roundabout way of firing the required trigger, but gives you the ability pass the trap's variable bindings, along with the trigger, to the second alarm. In addition, Send Trap enables an alarm being managed by one NerveCenter server to communicate with an alarm being managed by another server, while Fire Trigger does not.

Of course, you aren't limited to sending traps to NerveCenter. You can send a trap to any application that knows how to process SNMP traps.

TO ADD A SEND TRAP ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Send Trap** from the pop-up menu.
The Send Trap Action dialog is displayed.

3. In the **Source** field, enter information about node whose address you want to appear in the agent-address field of the trap PDU.

The valid values for this field are:

- **\$NodeName** (the default value), which represents the node associated with the trigger that caused the transition.
- **\$NCHostName**, which represents the node on which the active NerveCenter server is running.
- A node name.
- An IP address. Using an IP address is generally more efficient than using a node name because it eliminates the name-to-address translation.

4. In the **Destination** field, enter information about the node to which the trap should be sent. The valid values for this field are the same as those for the Source field. \$M is the default.
5. Enter in the **Port** field the number of the port on the destination machine to which the trap should be sent. Generally, SNMP traps are received on port 162, so 162 is the default value.
6. Enter a community name in the **Community** field. This is the community name that a manager needs to know in order to access the agent that is sending the trap. The default value is public.
7. Select one of the three options from the **Trap Numbers** drop-down list: **Default**, **Trap**, and **Custom**.
 - If you select Default, your trap's generic trap number will be 6, and its specific trap number will be 1.
 - If you select Trap, your trap's generic and specific trap numbers will match those of the trap associated with the trigger that caused the alarm transition.
 - If you select Custom, you can specify a generic trap number using the Generic drop-down list. In addition, if you select a generic trap number of 6, you can enter a specific trap number in the Specific field.
8. In the **Enterprise** field, enter an object identifier, or the corresponding name, for the device that is the source of the trap. The valid values for this field are:
 - \$P (the default), which indicates that the enterprise field in the trap you're sending should match the enterprise field in the trap associated with the trigger that caused the alarm transition.
 - Note that if the trigger that caused the transition with which this action is associated is not caused by a trap, \$P will not have a value, and the Send Trap action will not take place.
 - An object identifier, such as 1.3.6.1.4.1.9.
 - A name associated with an object identifier in an ASN.1 file.

Caution: Be aware that traps from the LogMatrix object ID (1.3.6.1.4.1.78) cannot be seen by NerveCenter because they are forwarded to your platform.

9. Enter information for each variable binding to be included in the trap PDU.

For each variable binding, perform the following steps.

- a. If you want a variable binding to contain exactly the same information as the corresponding variable binding in the trap associated with the trigger that caused the alarm transition, select **\$P** from the **Base Object** list and then select the **Insert** button

If you perform [Step a](#), you can skip the remaining steps in this procedure. Otherwise, go on to [Step b](#).

- b. Select a base object from the **Base Object** list.
- c. Select an attribute from the **Attribute** list.
- d. Type an instance in the **Instance** field.

Using your base object, attribute, and instance, NerveCenter creates the object identifier portion of the variable binding. For example, if you supply the base object system, the attribute sysUpTime, and the instance 0, NerveCenter builds an OID of 1.3.6.1.2.1.1.3.0.

- e. Enter a value for the attribute instance in the **Attribute Value** field.
- f. Select the **Insert** button.

10. Select the **OK** button in the Send Trap Action dialog.
 11. Select the **OK** button in the Transition Definition window.
 12. Select the **Save** button in the Alarm Definition window.
-

Set Attribute

The Set Attribute alarm action enables you to set selected attributes of an alarm, a mask, a poll, or a node. For alarms, masks, and polls, you can turn an object on or off. For nodes, you can assign the node a property group, or you can suppress or unsuppress the node.

A good example of the use of this action occurs in the predefined alarm DwnStrmSnmpStatus, which is part of a behavior model that suppresses alarms from nodes that are downstream from a router that is down. The state diagram for this alarm is shown in [Figure 52](#).

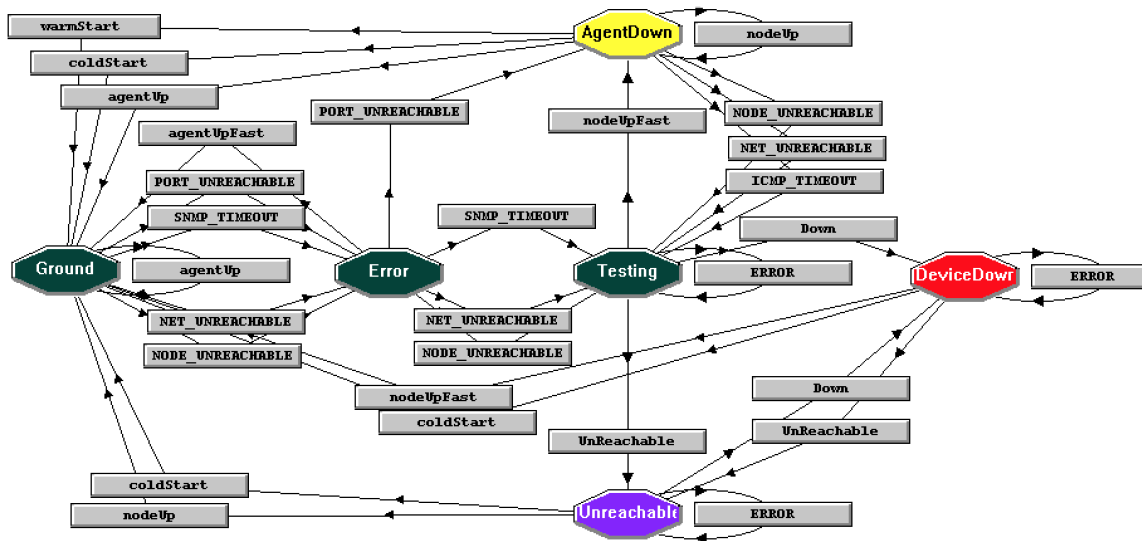


Figure 52: DwnStmSnmStatus Alarm

When the behavior model discovers that a node is unreachable because of a router that is down, it fires the trigger `Down` and uses the Set Attribute action to turn suppression on for the node it is tracking. Suppressing the node causes all insuppressible polls to stop polling the node. Similarly, if the poll `IcmpPoll` or `IcmpFastPoll` (both of these polls are insuppressible) determines that the node is reachable again, the alarm uses the Set Attribute action to turn suppression off for the node. At this point, normal polling resumes.

Note: If your Set Attribute alarm action turns an alarm off, any pending triggers fired by that alarm are cleared if the **Clear Triggers for Reset To Ground or Off** checkbox is checked in the alarm's definition window.

TO ADD A SET ATTRIBUTE ALARM TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **Set Attribute** from the pop-up menu.
The Set Attribute Action dialog is displayed.

The screenshot shows a dialog box titled "Set Attribute Action" with a question mark icon and a close button (X). The dialog contains the following fields:

- Object Type:** A dropdown menu with "Node" selected.
- Name:** A dropdown menu with "\$NODE" selected.
- Attribute:** A dropdown menu with "Suppress" selected.
- Value:** A dropdown menu with "On" selected.

At the bottom of the dialog are three buttons: "OK", "Cancel", and "Help".

In this release of NerveCenter, the Host field is not used.

3. From the **Object Type** drop-down list, select the type of object for which you want to set an attribute.
 4. Select the name of the object whose attribute you want set from the **Name** drop-down list.
For an alarm, a mask, or a poll, your options include all the objects of that type in the NerveCenter database. For a node, you can select any of the nodes in the NerveCenter database or the variable \$nodeName. This variable contains the name of the node associated with the trigger that caused the transition.
 5. Select the object attribute you want to set using the **Attribute** drop-down list.
If the Object Type is Alarm, Mask, or Poll, the Attribute field is read only because the only attribute you can set is State (the object's Enabled status). For a node, you can select either Property Group or Suppress.
 6. Select the value to which you want to set the attribute from the **Value** drop-down listbox.
 7. Select the **OK** button in the Set Attribute Action dialog.
 8. Select the **OK** button in the Transition Definition window.
 9. Select the **Save** button in the Alarm Definition window.
-

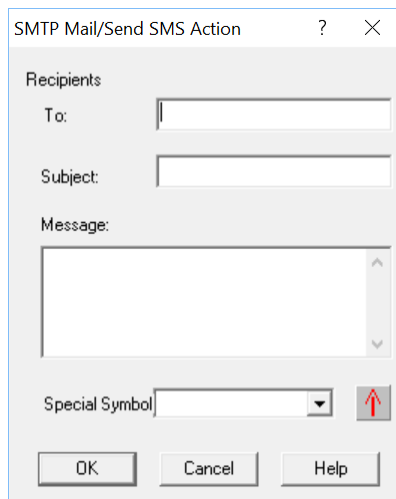
SMTP Mail

The SMTP Mail alarm action enables an alarm to send email concerning a transition.

Note: Before you can use this action, NerveCenter must specify an SMTP server. This setup is covered in [Specifying an SMTP Server for Mail Notification](#) in *Managing NerveCenter*.

TO ADD AN SMTP MAIL ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **SMTP Mail** from the selection control.
The SMTP Mail/Send SMS Action dialog is displayed.



3. Enter a recipient for the mail in the **To:** field.
4. Enter a subject line for the mail in the **Subject:** field.
5. Enter the message body in the **Message:** field.
The message text may be used as boilerplate wherein you insert a supported variable in the text, which is replaced at runtime with values taken from the live context.

To add a variable, position the cursor in the text and select a variable from the **Special Symbol** list. Then click the 'up' arrow to insert the selected variable into the text.
6. Select the **OK** button.
The new action appears in the list of actions in the Transition Definition window.

7. Select the **OK** button in the Transition Definition window.
8. Select the **Save** button in the Alarm Definition window.

SNMP Set

The SNMP Set alarm action enables you to set one or more values in the MIB of an SNMP agent residing on one of your managed nodes. When the transition with which this action is associated occurs, NerveCenter sends an SNMP set request, which includes information you've supplied, to the node where the agent resides.

TO ADD AN SNMP SET ACTION TO A TRANSITION

1. From the Transition Definition window, select the **New Action** list.
A list of available actions is displayed.
2. Select **SNMP Set** from the pop-up menu.
The SNMP Set Action window is displayed.

SNMP Set Action

Destination Host / IP Address: \$NODE

Community String: \$WRITE_COMMUNITY

Port: \$PORT

Variable Bindings

Base	Attribute	Instance	Value

Insert Update Delete Delete All

Attribute Value: Instance: \$I

Base Object: aal5VccEntry, acEntry, addressMap, addressMapControlEntry, addressMapEntry, alarmEntry, alHostEntry

Attribute:

OK Cancel Help

3. Enter the destination for the SNMP set request in the **Destination Host/IP Address** field, or leave the default value, \$NODE.

The valid values for this field are:

- \$NODE, a variable that contains the node associated with the trigger that caused the alarm transition. For example, if a poll generates the trigger, \$NODE contains the name of the node that was polled.
- The name of a node.
- The IP address of a node.

4. Enter a write community string in the **Community String** field, or leave the default value, \$WRITE_COMMUNITY.

The valid values for this field are:

- \$WRITE_COMMUNITY, a variable containing the write community value associated with the destination node.
- A community name.

5. Enter a port number in the **Port** field, or leave the default value, \$PORT. This field indicates the port to which the SNMP message will be sent.

The valid values for this field are:

- \$PORT, a variable containing the port number associated with the destination node. If the node's Port attribute is blank, \$PORT represents the value 161.
- A port number.

6. Build a list of variable bindings to be included in your set request's PDU (protocol data unit). Each variable binding specifies an attribute to be set and the value to which it should be set.

For each variable binding you want to add to the PDU, perform these steps:

- a. Select a base object from the **Base Object** list.

The base object list contains all the base objects referred to in your compiled MIB. Once you select a base object, the attributes of that object are listed in the Attribute list.

- b. Select an attribute from the **Attribute** list.

- c. Type a value for your attribute in the **Attribute Value** field.

- d. Specify which instance of the attribute you want to set using the **Instance** field.

If the attribute is a zero-instance attribute, NerveCenter automatically supplies the instance (0) when you insert the variable binding into the Variable Binding list. In addition, NerveCenter provides a variable, \$I, that you can use to refer to instance information in the poll or trap mask that generated the trigger.

- e. Select the **Insert** button.

Your variable binding is appended to the Variable Binding list.

Note: The SNMP Set Action dialog also enables you to modify and delete existing variable bindings. Use the Update, Delete, and Delete All buttons for these operations.

7. Select the **OK** button in the SNMP Set Action dialog.
The new action is added to the Actions list in the Transition Definition window.
 8. Select the **OK** button in the Transition Definition window.
 9. Select the **Save** button in the Alarm Definition window.
-

Performing Actions Conditionally (Action Router)

When an alarm transition occurs, all the actions associated with that transition are performed unconditionally. However, the responsibility of one action—Action Router—is to send information about the transition to the Action Router facility, which performs actions *conditionally*. That is, the Action Router action always takes place, but the Action Router facility may or may not initiate some other action.

Whether the Action Router facility performs one or more actions—such as executing a command or logging data to a file—depends on rules that you’ve set up using the Action Router. For example, you might want to specify that if a particular alarm transition occurs at night or on the weekend, an administrator should be paged. In this case, the alarm transition has the Action Router action associated with it, and the Action Router rule looks like this:

```
$DayOfWeek >= MONDAY and $DayOfWeek <= FRIDAY and ($Time < 08:00  
or $Time > 17:00) or ($DayOfWeek == SATURDAY or $DayOfWeek ==  
SUNDAY)  
-> Paging 5551234567:911:#
```

All actions that can be performed from an alarm transition can be performed from the Action Router, except for the Alarm Counter and Action Router actions. Also, rule conditions can be built using many types of data, for example:

- The name of an alarm. Did the transition take place in an instance of this alarm?
- The name of a node. Was the alarm instance in which the transition took place monitoring this node?
- The name of a property group. Does the node that was being monitored have this property group?
- The severity of the destination alarm state.
- The name of the trigger that caused the transition.

For a complete list of the variables that can be used in an Action Router rule condition, see the table [NerveCenter Variables on page 301](#).

The remainder of this chapter explains how to determine what Action Router rules have already been defined and how to create new rules.

Listing Existing Action Router Rules

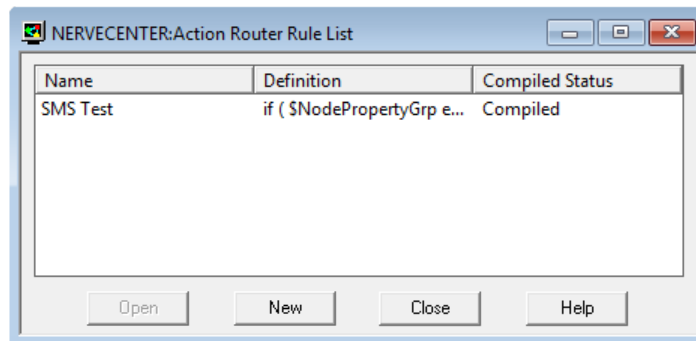
This section explains how to display a list of the Action Router rules currently defined in the NerveCenter database. The section also explains how to view the definition of a particular rule.

For information on creating a new rule, see [Creating an Action Router Rule on page 318](#).

TO DISPLAY A LIST OF ACTION ROUTER RULES AND THEN DISPLAY A PARTICULAR RULE'S DEFINITION

1. From the client's **Admin** menu, choose **Action Router Rule List**.

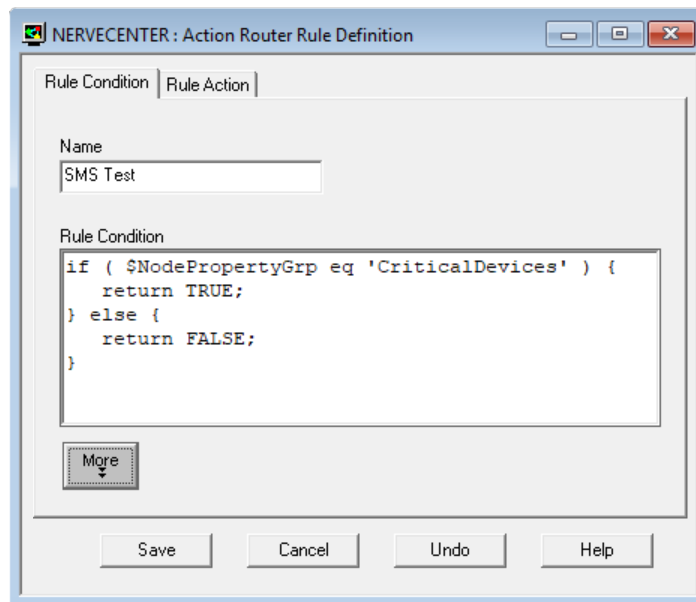
The Action Router Rule List window is displayed.



This window lists all currently defined Action Router rules. If enough room is available in the window, you can see, for each rule, the condition under which actions will be performed (the rule condition) and the actions that will be performed under those conditions (the rule actions).

If you can only see part of the rule, you can either enlarge the window or perform the following steps.

2. Double-click the rule whose definition you want to see.
The Action Router Rule Definition window is displayed.



3. Select the **Rule Condition** tab to see the rule condition and the **Rule Action** tab to see the actions defined for the rule.

In the figure above, the condition says, “If the alarm transition occurs after hours on a week day or on a weekend, take the actions listed on the Rules Action page.”

Creating an Action Router Rule

There are two components to an Action Router rule: a condition and a list of actions. For example, suppose you need to develop a rule that will cause NerveCenter to send you e-mail if a device goes down. The rule's condition might be:

```
$TriggerName eq "deviceDown"
```

This means that you want to know if the Action Router is notified of a transition that occurred as a result of a deviceDown trigger.

The rule's action might be:

```
SMTP Mail networkadmin@yourcompany.com
```

This means that if the condition is met, NerveCenter should send SMTP mail to the address shown.

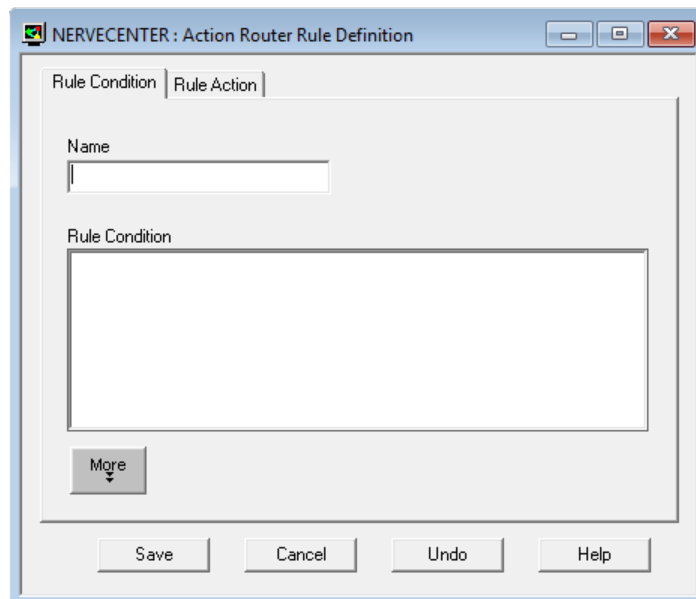
The next two sections explain how to create such rule conditions and rule actions; note that you must create both a condition and one or more actions to complete an Action Router rule.

Defining a Rule Condition

Defining a rule condition is one part of defining an Action Router rule. After defining the rule condition, you must define a rule action to complete the Action Router rule. For information on defining a rule action, see the section [Defining a Rule Action on page 323](#).

TO DEFINE A RULE CONDITION

1. From the client's **Admin** menu, choose **Action Router Rule List**.
The Action Router Rule List window is displayed.
2. Select the **New** button in the Action Router Rule List window.
The Action Router Rule Definition window is displayed.



3. Enter a unique name for your Action Router rule in the **Name** field.

Note: The maximum length for Action Router rule names is 255 characters.

4. Write your rule condition in the Rule Condition text area.

You write this rule condition using Perl. However, you need not write a complete Perl statement. You can assume the following context:

```
if (...) {  
    ruleAction;}  
}
```

All you must supply is the condition that would fit inside the parentheses. For example, **\$OriginStateSev eq "Normal"** is a complete rule condition.

To help you write rule conditions, NerveCenter provides several aids:

- A set of variables that contain data you can use in your rule condition. We've already seen a number of these, such as `$DayOfWeek`, `$Time`, and `$OriginStateSev`. For a complete list of the variables available to you, see the section [NerveCenter Variables on page 301](#).
- A set of functions that you can use in your rule conditions. These functions enable you to determine whether a variable contains a substring, to access information in the variable bindings of a trap that caused an alarm transition, and more.

For more information about these functions, see the section [Functions for Use in Action Router Rule Conditions on the facing page](#).

- A pop-up menu that lists the variables and functions you can use in a rule condition and enables you to enter the name of a variable or function in the rule-condition editing area. For further information about this pop-up menu, see the section [Using the Pop-Up Menu for Perl on page 182](#).
- Lists of the alarms, days, nodes, properties, property groups, severities, and triggers that you can use in a rule condition. Selecting an item from one of these list writes the name of the selected object to the rule-condition editing area.

For further information about these lists, see the section [Using Action Router Object Lists on the facing page](#).

Note: When creating poll conditions, trigger functions, and Perl subroutines, you can choose between using multiple Perl interpreters or a Global Perl interpreter. If you configure your poll conditions, trigger functions, and Perl subroutines to use the Global Perl interpreter, Action Router rules that take a long time to run, such as logging to a file, performing database queries, or issuing external system calls, can slow down NerveCenter's performance. If you have need of such Perl scripts in your environment, use the scripts sparingly. See [NerveCenter and Perl on page 45](#) for more information.

Once you've finished building your rule condition, you must go to the Rule Action page and build a list of rule actions. For instructions on how to build this list, see the section [Defining a Rule Action on page 323](#).

Functions for Use in Action Router Rule Conditions

NerveCenter provides a number of functions that you can use in your Action Router rule conditions. The list below indicates what types of functions are available and where you can find detailed information about each function:

- **Variable-binding functions.** These functions enable you to determine the number of variable bindings in a trigger's variable-binding list and to obtain information about each variable binding. For instance, you can retrieve the subobject and attribute associated with a variable-binding and the value of a variable-binding.

For reference information about these functions, see the section [Variable-Binding Functions on page 204](#).

- **String-matching functions.** These functions enable you to determine whether a string contains another string or a particular word. The functions are useful in conditions that test the value of a variable or variable binding for a substring.

For reference information about these functions, see the section [String-Matching Functions on page 181](#).

- **in().** This function determines whether one scalar value is in a set of scalar values.

For reference information about this function, see the section [in\(\) Function on page 179](#).

- **Counter().** This function returns the current value of an alarm counter. For reference information about this function, see the section [Counter\(\) Function on page 299](#).

- **NC::AlarmCounters.** This function enables you to do any of the following: increment alarm counters by a number other than one, decrement alarm counters by a number other than one, create alarm counters, set alarm counters, and retrieve alarm counters. For reference information about this function, see the section [NC::AlarmCounters on page 179](#).

NC::AlarmCounters are independent of and not related to the alarm action Alarm Counter. See [Alarm Counter on page 271](#) for details.

Using Action Router Object Lists

If you are writing an Action Router rule condition and need to enter the name of an alarm, you do not need to:

- Look up the name of the alarm in the Alarm Definition List window.
- Type the name of the alarm in the Rule Condition editing area.

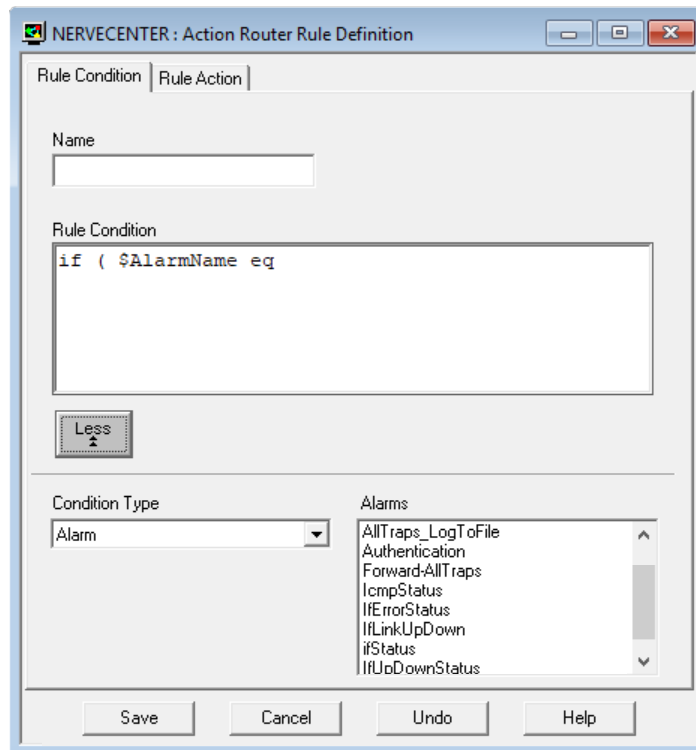
Instead, you can select the name of the alarm from a list of alarms on the Rule Condition page. Selecting this name copies the name to the Rule Condition editing area, at the point of the cursor.

In addition to a list of alarms, the Rule Condition page provides lists of:

- Days (Days are not really NerveCenter objects.)
- Nodes
- Properties
- Property groups
- Severities
- Triggers

HOW TO ENTER THE RULE CONDITION `$ALARMNAME EQ 'AUTHENTICATION'`

1. In the Rule Condition editing area enter the text `$AlarmName eq` using the editing area's pop-up help menu or your keyboard.
2. Select the **More** button on the Rule Condition page to expand the page.



3. Select **Alarm** from the **Condition Type** drop-down list.

The list to the right of the drop-down list is populated with the names of all the alarms in the NerveCenter database.

Note: If you were writing a different rule condition, you could have selected a different object from the drop-down list.

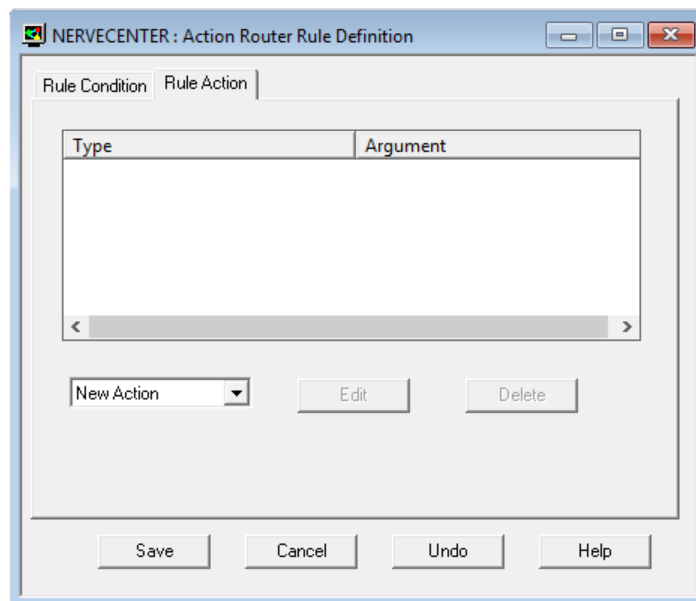
4. Double-click **Authentication** in the **Alarms** list.
This action causes the text '**Authentication**' to be added to the rule condition.
5. After you've defined your rule's action, select the **Save** button.

Defining a Rule Action

Once you've created an Action Router rule condition, as described in the section [Defining a Rule Condition on page 319](#), you must create a rule action to complete your Action Router rule. This action rule contains descriptions of one or more actions that you want to be performed when the rule condition is met.

TO CREATE AN ACTION RULE

1. In the Rule Composition window, select the **Rule Action** tab.
The Rule Action page is displayed.



2. Select the **New Action** list.

A list of available actions is displayed. Except for the Action Router and Alarm Counter actions, you can add to the rule any action that you can perform from an alarm transition.

Actions are described in "[Alarm Actions](#)" on page 269.

3. Select an action from the list.

All of the actions except Delete Node and Notes require parameters, so a dialog box appears.

Again, refer to the appropriate section in "[Alarm Actions](#)" on page 269 for an explanation of how to supply the necessary parameters.

4. Repeat [Step 2](#) and [Step 3](#) for each action that you want to add to the rule action.
 5. Select the **Save** button at the bottom of the Rule Composition window.
-

Most behavior models employ only one alarm. However, some models require two or more alarms. If a model uses more than one alarm, the alarms generally communicate using the Fire Trigger alarm action. That is, one alarm fires a trigger that causes a transition in a second alarm.

This chapter presents an example of a multi-alarm behavior model (sometimes referred to as multi-tier behavior models), which might serve as an example for your own models.

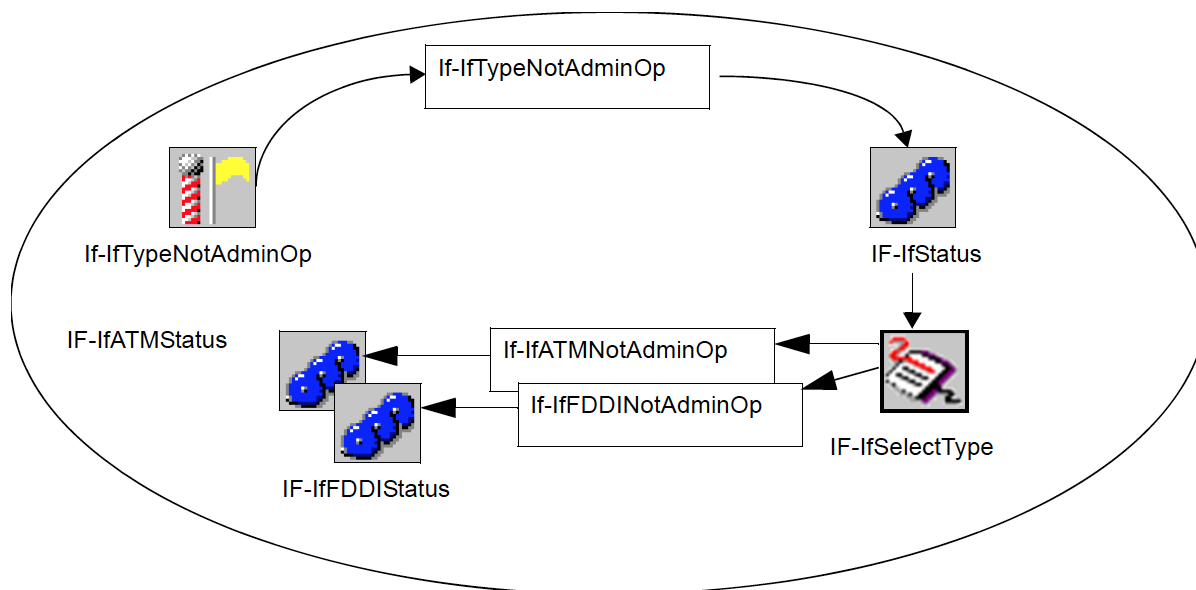
Note: Another good example of a multi-alarm behavior model is the downstream alarm suppression model, `NodeStatusDwnStrm`, that ships with NerveCenter. For more information, refer to "[Downstream Alarm Suppression](#)" on page 383.

IfUpDownStatusByType

IfUpDownStatusByType is one of the multi-alarm behavior models shipped with NerveCenter and provides interface management for devices that can be managed using the MIB-II and Frame Relay MIBs. This model manages the following types of interfaces:

- Asynchronous Transfer Mode (ATM)
- Integrated Services Digital Network (ISDN)
- Fiber Distributed Data Interface (FDDI)
- Frame Relay Permanent Virtual Circuit (PVC) subinterfaces
- Frame Relay
- Local Area Network (LAN)
- Switched Multimegabit Data Service (SMDS)
- Synchronous Optical Network (SONET)
- Wide Area Network (WAN)

The majority of the alarms in this model are subobject scope alarms that categorize an interface (the possible categories are listed above) and then monitor its status. For most interfaces, the interface can be up, down, or in testing mode. (The exception is a Frame Relay PVC, which can only be up or down.)



When an alarm instance transitions to one of these states, it executes an Inform action to notify a network management platform of the new state. For this Inform action to have the desired effect, you must integrate the **trapd.conf.txt** file supplied with these models with the standard NerveCenter **trapd.conf**. The **trapd.conf.txt** file along with the .mod file resides in the **/model/interface_status/updown_bytype directory**. For information about importing behavior models into NerveCenter, see [Importing Node, Object, and Behavior Model Files on page 364](#).

The interface status alarms are listed below:

- IF-IfATMStatus
- IF-IfFDDIStatus
- IF-IfFramePVCStatus
- IF-IfFrameRelayStatus
- IF-IfISDNStatus
- IF-IfLANStatus
- IF-IfSMDSSStatus
- IF-IfSonetStatus
- IF-IfWANStatus

The model file also includes three other alarms: IF-IfStatus, IF-IfColdWarmStart, and IF-IfNmDemand.

IF-IfStatus Alarm

The predefined alarm IF-IfStatus is a subobject scope alarm that monitors interfaces on the network. Its definition is shown in Figure 53.

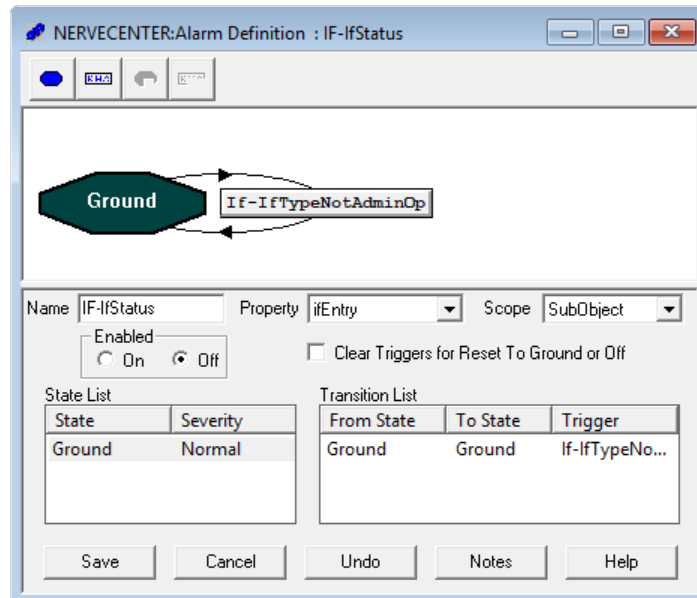


Figure 53: IF-IfStatus Alarm

IF-IfStatus listens for the trigger IF-IfTypeNotAdminOp, which is fired whenever an interface is not operationally up (either down or in testing mode). When IF-IfStatus transitions to IF-IfTypeNotAdminOp, the alarm fires a Perl subroutine, IF-SelectType.

IF-SelectType Perl Subroutine

IF-SelectType is a Perl subroutine composed of an If statement that reads the instance of ifEntry.ifType to determine the interface type being monitored and to fire the appropriate trigger.

```

Name: IF-SelectType
Subroutine:
if (ifEntry.ifType == 6) { FireTrigger("If-ifLANNNotAdminOp"); } # 802.2 interface
elsif (ifEntry.ifType == 7) { FireTrigger("If-ifLANNNotAdminOp"); } # 802.3 interface
elsif (ifEntry.ifType == 15) { FireTrigger("If-ifFDDINotAdminOp"); } # FDDI interface
elsif (ifEntry.ifType == 17) { FireTrigger("If-ifWANNotAdminOp"); } # SDLC interface
elsif (ifEntry.ifType == 18) { FireTrigger("If-ifWANNotAdminOp"); } # DSL interface
elsif (ifEntry.ifType == 19) { FireTrigger("If-ifWANNotAdminOp"); } # EL interface
elsif (ifEntry.ifType == 20) { FireTrigger("If-ifbISDNNotAdminOp"); } # bISDN interface
elsif (ifEntry.ifType == 21) { FireTrigger("If-ifWANNotAdminOp"); } # pISDN interface
elsif (ifEntry.ifType == 22) { FireTrigger("If-ifWANNotAdminOp"); } # pPPPS interface
elsif (ifEntry.ifType == 30) { FireTrigger("If-ifWANNotAdminOp"); } # DS3 interface
elsif (ifEntry.ifType == 31) { FireTrigger("If-ifSMDsNotAdminOp"); } # SIP interface
elsif (ifEntry.ifType == 32) { FireTrigger("If-ifFRAMNotAdminOp"); } # fRELAY interface
elsif (ifEntry.ifType == 37) { FireTrigger("If-ifATHNotAdminOp"); } # ATM interface
elsif (ifEntry.ifType == 39) { FireTrigger("If-ifSONETNotAdminOp"); } # SONET interface
elsif (ifEntry.ifType == 43) { FireTrigger("If-ifSMDsNotAdminOp"); } # smsdDxl interface
elsif (ifEntry.ifType == 44) { FireTrigger("If-ifFRAMNotAdminOp"); } # fRelaySERVICE
elsif (ifEntry.ifType == 45) { FireTrigger("If-ifWANNotAdminOp"); } # V35 interface
elsif (ifEntry.ifType == 46) { FireTrigger("If-ifWANNotAdminOp"); } # HSSI interface
elsif (ifEntry.ifType == 47) { FireTrigger("If-ifWANNotAdminOp"); } # HIPPI interface
elsif (ifEntry.ifType == 50) { FireTrigger("If-ifSONETNotAdminOp"); } # sonetPATH interface
  
```

Figure 54: IF-SelectType Perl Subroutine

IF-SelectType fires the appropriate trigger to instantiate the correct interface-type alarm for the interface that is in a non-operational status.

Interface-type Alarms

The IfUpDownStatusByType behavior model has an alarm for each interface type (ATM, ISDN, FDDI, and so on) that it monitors. The interface alarms (with the exception of IF-IfFramePVCStatus) are identical. The definition for these alarms, is shown in Figure 55.

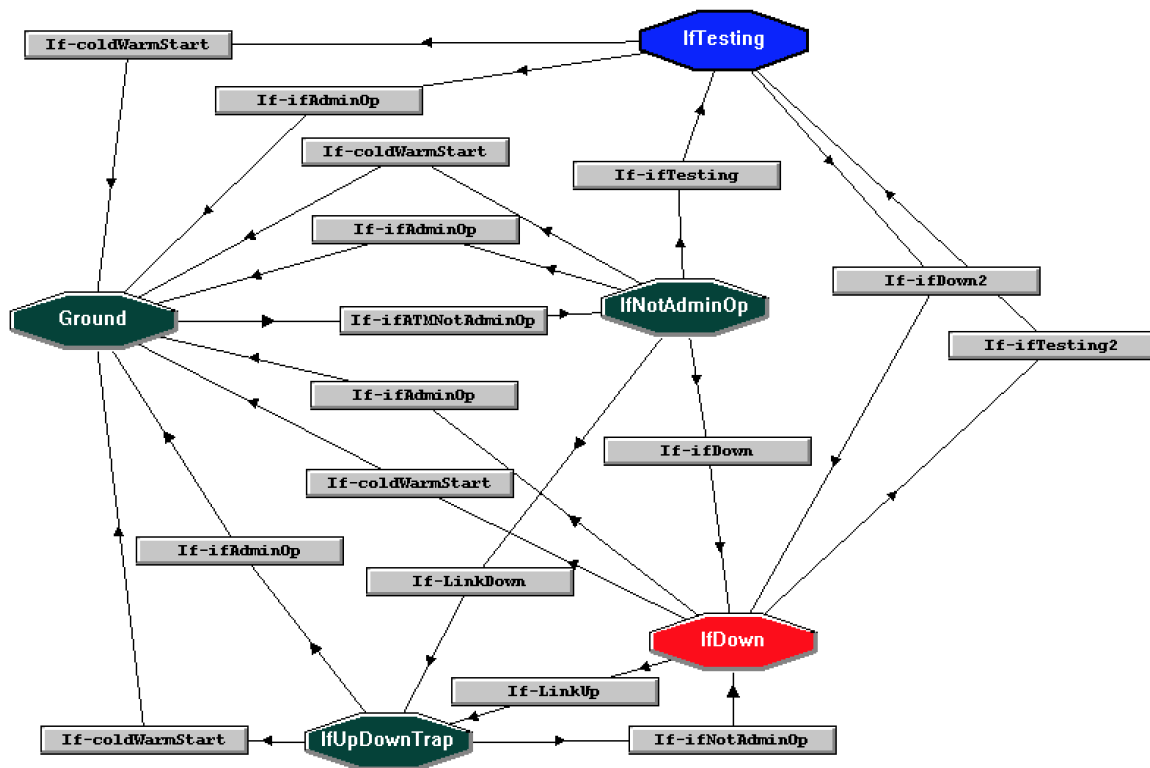


Figure 55: Interface-type Alarms State Diagram

The interface-type alarms (with the exception of IF-IfFramePVCStatus) contain the following states:

- **Ground** - No evidence that the interface is down, or in testing mode.
- **IfNotAdminOp** - An initial indication that an interface is either down or in testing mode has been received by a poll. The interface is categorized (ATM, FDDI, LAN, and so on) and the appropriate alarm is transitioned.
- **IfUpDownTrap** - Mask indicates that a link is either up or down. The interface is polled. If the interface is up, NerveCenter sends a 1512 Inform to the platform and returns to Ground. If a cold or warm start is detected, returns to Ground. If the interface is down or in testing, NerveCenter sends a 1514 Inform to the platform and goes to IfDown.
- **IfDown** - Poll indicates that an interface down. NerveCenter sends a 1514 Inform to the platform. The interface is polled. If the interface is up, NerveCenter sends a 1512 Inform to the platform and returns to Ground. If a cold or warm start is detected, returns to Ground. If the interface is in some test mode, NerveCenter sends a 1513 Inform to the platform and goes to IfTesting.
- **IfTesting** - Poll indicates that an interface is in some test mode. NerveCenter sends a 1513 Inform to the platform. The interface is polled. If the interface is up, NerveCenter sends a 1512 Inform to the platform and returns to Ground. If a cold or warm start is detected, returns to Ground. If the interface is down, NerveCenter sends a 1514 Inform to the platform and goes to IfDown.

IF-IfFramePVC

Unlike the other interface-type alarms, the IF-IfFramePVC relies on a frame relay MIB with which to monitor frame relay permanent virtual circuit (PVC) subinterfaces. NerveCenter instantiates IF-IfFramePVC when a frame relay PVC interface is non-active. The definition for IF-IfFramePVC, is shown in Figure 56.

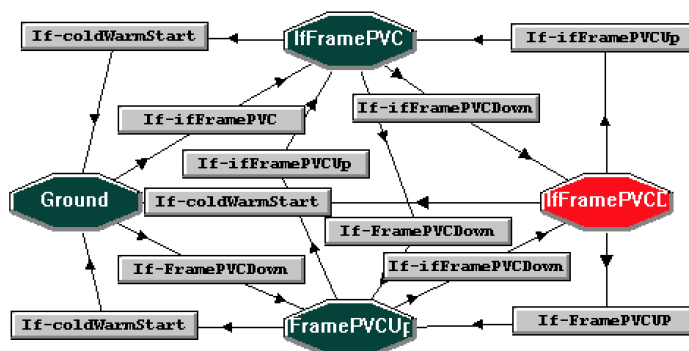


Figure 56: IF-IfFramePVC State Diagram

If-IfFramePVCStatus contains the following states:

- **Ground** - No evidence that the interface is down. If the interface is down, goes to FramePVCUp/Down. If the interface is active, goes to IfFramePVC.
- **FramePVCUp/Down** - Mask indicates that a link is either up or down. The interface is polled. If the interface is up, NerveCenter sends a 1510 Inform to the platform and returns to IfFramePVC. If a cold or warm start is detected, returns to Ground. If the interface is down, NerveCenter sends a 1511 Inform to the platform and goes to IfFramePVCDown.
- **IfFramePVCDown** - Poll indicates that an interface down. NerveCenter sends a 1511 Inform to the platform. The interface is polled. If the interface is up, NerveCenter sends a 1510 Inform to the platform and goes to IfFramePVC. If a cold or warm start is detected, returns to Ground. If the interface is up or down, goes to FramePVCUp/Down.
- **IfFramePVC** - Interface is active. If a cold or warm start is detected, returns to Ground. If the interface is up or down, goes to FramePVCUp/Down. If the interface is down, NerveCenter sends a 1511 Inform to the platform and goes to IfFramePVCDown.

IfColdWarmStart Alarm

The IfColdWarmStart alarm detects that a device has been restarted and fires a trigger that causes all the interface-type alarms monitoring that device to return to Ground state.

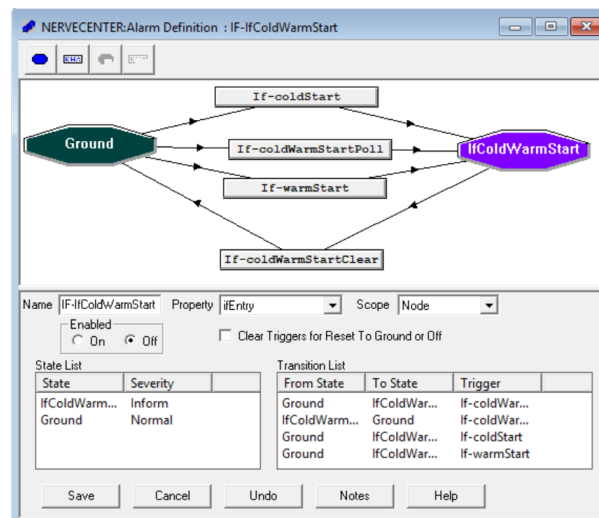


Figure 57: IF-IfColdWarmStart Alarm

The IfColdWarmStart alarm also fires a trigger that causes a transition in an IfNmDemand alarm.

IfNmDemand Alarm

An IfNmDemand alarm is instantiated whenever an interface-type alarm transitions to the up, down, testing, or ground state.

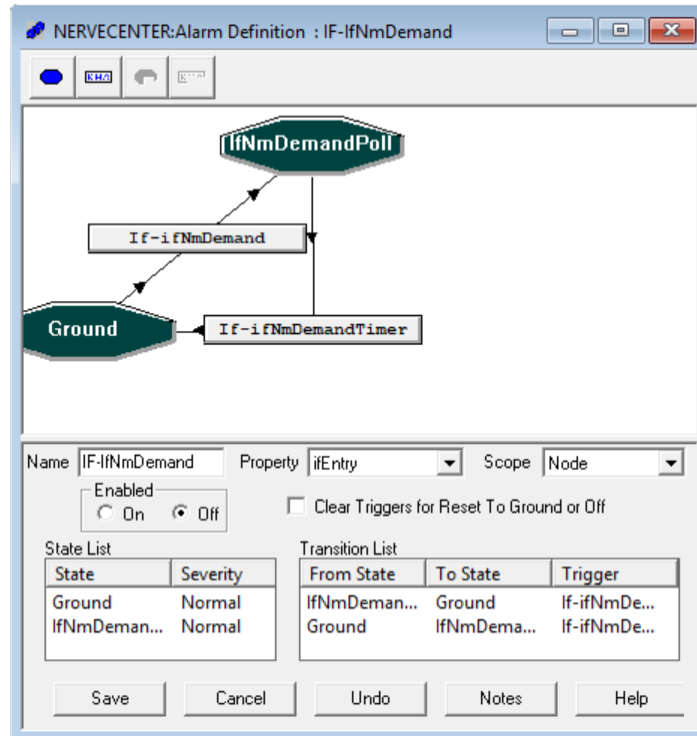


Figure 58: IF-IfNmDemand Alarm

When the alarm is created and transitions to the IfNmDemandPoll state, it executes an Inform action that causes your network management platform to demand poll the appropriate device and reflect the current state of the device and its interfaces in its topology maps. The Inform action that requests the demand poll is made outside of the status alarms—in a node-scope alarm—to help cut back to the number of requests that can be sent to your network management platform.

The majority of this book has discussed the function of the various NerveCenter objects and how to create those objects.

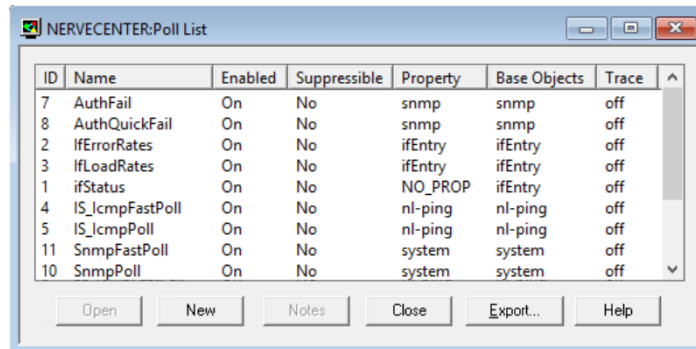
This chapter discusses how to perform other operations on objects, such as copying and deleting them. It also covers how to change selected object attributes without returning to the object definition windows. For example, the chapter explains how to change an alarm's property without returning to the Alarm Definition window.

Enabling Objects

As we've mentioned many times, a behavior model does not become functional until all of the polls, masks, and alarms in the model are enabled. This section explains how you can quickly enable, or disable, any poll, trap mask, OpC mask, or alarm.

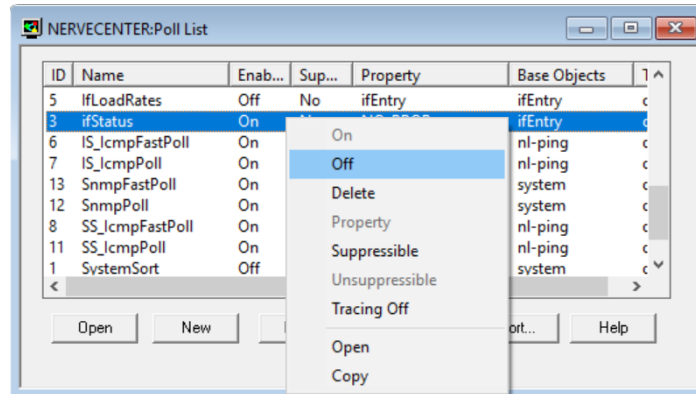
TO ENABLE ONE OF THESE OBJECTS

1. Open the appropriate list window: the Poll List, Mask List, OpC Mask List, or Alarm Definition List window. In this example, the Poll List window.



2. Select the object whose enabled status you want to change.

- With your cursor positioned over the selected object, click the right mouse button to display a pop-up menu listing actions you can perform against the object.



If the object is disabled, the Off entry will be grayed out, and if the object is enabled, the On entry will be grayed out.

- Select **On** from the menu to enable the object, or **Off** to disable it.
The object is now enabled. It's not necessary to save this change in order for it to take effect.

Copying Objects

Being able to copy objects can be very convenient. For example, if you want to create a property group that is exactly the same as an existing one except that it contains one additional property, it's nice to be able copy the existing property group, give the copy a name, and add the one property—instead of creating a new property group and adding a long list of properties to it. The same is true if you need to create a new alarm that is similar to an existing alarm, or a new poll that is similar to an existing one.

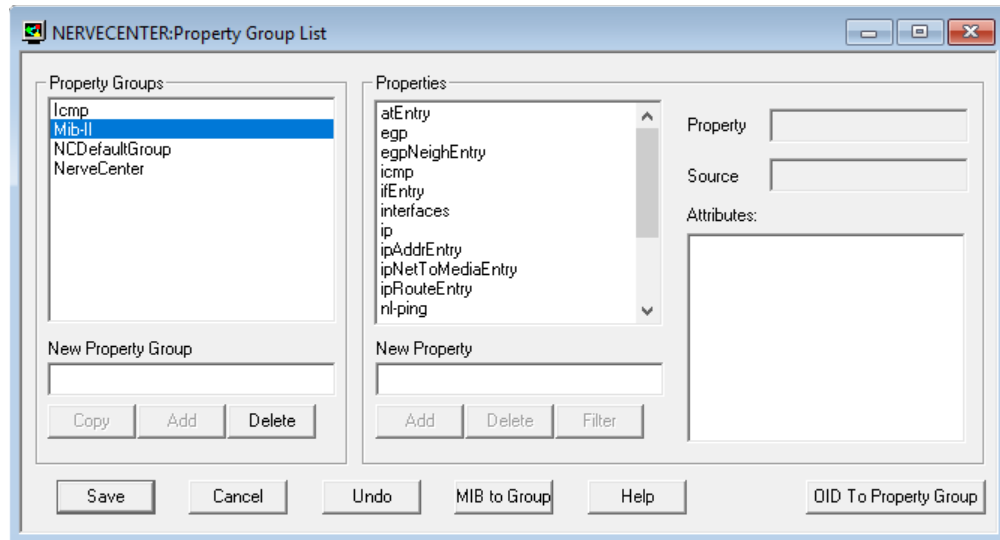
NerveCenter enables you to copy most objects. To copy a property group, you select a Copy button in the Property Group List button. To copy any other object (that supports a copy operation), you select Copy from a pop-up menu associated with the object.

Copying a Property Group

This section explains how to create a copy of an existing property group.

TO COPY A PROPERTY GROUP

1. Open the Property Group List window.



2. Select the property group you want to copy from the **Property Group** list.
3. Enter a name for the copy of the property group in the **New Property Group** field.
The Copy button is enabled.
4. Select the **Copy** button.
5. Select the **Save** button.

You now have an exact copy of the property group you began with. You'll probably want to add properties to, or remove properties from, the new property group and save it again.

Copying Other Objects

You can easily copy Alarms, Polls, Masks, Nodes, Action Router rules, and Perl subroutines.

TO COPY ONE OF THESE OBJECTS

1. Open the appropriate list window.
2. Select the object you want to copy from the list.
3. With your cursor over the selected object, click the right mouse button to display a pop-up menu of actions you can perform against the object.
4. Select **Copy** from the pop-up menu.
A definition window is displayed. The window contains a complete definition except for a name.
5. In the definition window, enter a name for the copied object.
6. Select the **Save** button in the definition window.

You now have an exact copy of the object you began with. Make any necessary changes to the copy, and save it again.

Deleting Objects

If you have objects in your NerveCenter database that you know you'll never use again, you can delete them.

There are two methods of deleting objects in NerveCenter:

- **Delete button:** Some objects — Property Groups, OID to Property Group mappings, and Severities — are [deleted via a button](#) in the appropriate definition window.
- **Delete pop-up:** Other objects — Alarms, Polls, Masks, Nodes, Action Router rules, and Perl subroutines — are [deleted via a pop-up menu](#) in the appropriate list window.

Using a Delete Button

This section explains how to delete a property group, an OID to property group mapping, or a severity.

TO DELETE ONE OF THESE OBJECTS

1. Open the appropriate list window.
2. Select from the list the object you want to delete.
A Delete button is enabled.

3. Select the **Delete** button.

A property group can not be deleted if it is currently assigned to a node or is being used in an OID to property group mapping. If you attempt to delete a property group that is being used in one of these ways, you'll see a warning message. Of course, you can remove the dependency and then delete the property group.

Similarly, you can't delete a severity that is being used in an alarm. If you try to do so, you see a dialog similar to the one shown in [Figure 59](#):

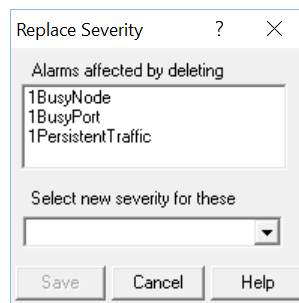


Figure 59: Replace Severity Dialog

The dialog in the figure indicates that the selected severity is being used in the alarm `SynBoardChannel`. If you want to go ahead and delete the severity, you must first change the severity of the affected state in this alarm. You do this by selecting a severity from the drop-down list and selecting the **Save** button. (You'll also have to confirm that you want to replace the severity.)

Using a Pop-Up Menu

This section explains how to delete an alarm, a poll, a mask, a node, an Action Router rule, or a Perl subroutine.

Caution: Before deleting an alarm, a poll, a mask, a node, an Action Router rule, or a Perl subroutine, make sure that it is not used in a behavior model and is not required to transition any alarms.

TO DELETE ONE OF THESE OBJECTS

1. Open the appropriate list window.
2. Select the object you want to delete.
3. With your cursor positioned over the selected object, click your right mouse button to display a pop-menu that lists actions you can perform from this window.

4. Select the **Delete** entry from the pop-up menu.
A Confirm Deletion dialog appears, asking if you're sure you want to delete the object.
 5. Select the **Yes** button in the Confirm Deletion dialog.
The object is deleted.
-

Changing an Object Property or Property Group

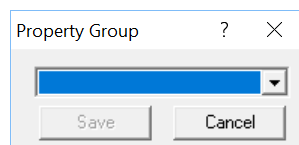
NerveCenter provides shortcuts for changing a poll's or an alarm's property and for changing a node's property group.

Changing Poll or Alarm Properties

This section explains how to change the property attribute of a poll or an alarm.

TO CHANGE AN OBJECT'S PROPERTY

1. Make sure that the poll's or alarm's enabled status is off.
For instructions on how to disable an object, see [Enabling Objects on page 333](#).
2. With the Poll List or Alarm Definition List window open, select the object whose property you want to change.
3. With your cursor positioned over the selected object, click your right mouse button to display a pop-menu that lists actions you can perform from the list window.
4. Select **Property** from the pop-up menu.
The Property dialog is displayed.



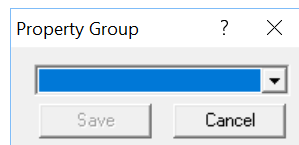
5. Select a new property for your object from the drop-down listbox in the Property dialog.
The Save button is enabled.
 6. Select the **Save** button.
The object's property is changed. Re-enable the object if necessary.
-

Changing a Node Property Group

This section explains how to change a node's property group without going to the Node Definition window.

TO CHANGE A NODE'S PROPERTY GROUP

1. Select Node List from the client's Admin menu.
The Node List window appears.
2. Select the node whose property group you want to change.
3. With your cursor positioned over the selected node, click your right mouse button to display a pop-up menu that lists the actions you can take from the Node List window.
4. Select Property Group from the pop-up menu.
The Property Group dialog is displayed.



5. Select the node's new property group from the drop-down listbox in the Property Group dialog.
The dialog's Save button is enabled.
6. Select the dialog's **Save** button.
The node's property group is changed.

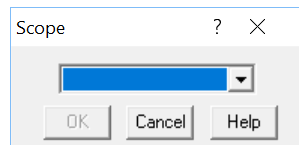
Changing an Alarm Scope

It's rarely necessary to change the scope of an alarm since determining the alarm's scope is usually a very fundamental part of designing the alarm. However, if the need to change an alarm's scope does arrive, you can make this change from the Alarm Definition List window.

TO CHANGE AN ALARM'S SCOPE

1. Choose **Alarm Definition List** from the client's **Admin** menu.
The Alarm Definition List window is displayed.
2. Select the alarm whose scope you want to change.
3. With your cursor positioned over the selected alarm, click your right mouse button to display a pop-up menu that lists the operations you can perform from the Alarm Definition List window.

4. Select **Scope** from the pop-up menu.
The Scope dialog appears.



5. Select a scope from the drop-down listbox in the Scope dialog.
The dialog's Save button is enabled.
6. Select the **Save** button.
The alarm's scope is changed.

Suppressing Polling

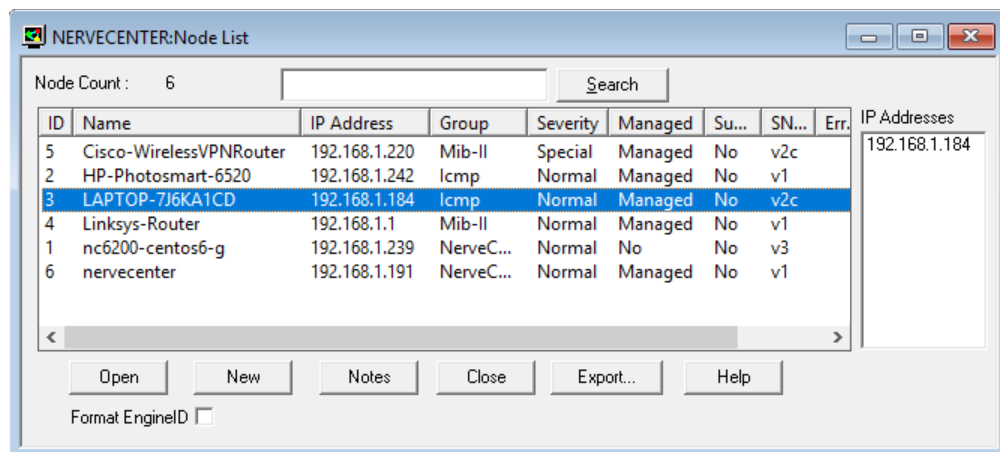
To prevent a particular poll from being sent to a particular node, the node must be suppressed, and the poll must be suppressible. By default, polls are suppressible; however, nodes are not. Therefore, keeping a poll from being sent to a node usually involves turning on the node's Suppressed attribute. You may have to edit the poll as well if someone has turned off its Suppressible attribute.

Suppressing a Node

This section explains how to suppress a node by enabling its Suppressed attribute.

TO ENABLE THIS ATTRIBUTE

1. From the client's **Admin** menu, select **Node List**.
The Node List window is displayed.



2. Select the node whose Suppressed attribute you want to enable.
3. With your cursor positioned over the selected node, right-click to display a pop-up menu.
4. Select **Suppress** from the pop-up menu.

This operation is the equivalent of checking the Suppressed in the Node Definition window.

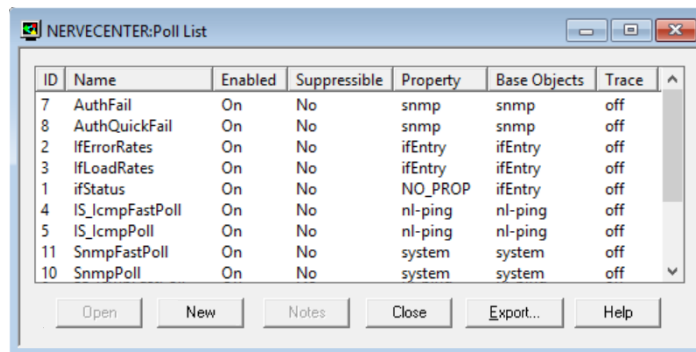
Making a Poll Suppressible

This section explains how to make a poll suppressible by enabling its Suppressible attribute.

By default, polls in NerveCenter are suppressible and poll only those nodes that are not suppressed. However, there may be specific polls that you want to occur on a node even when it is suppressed. In this case, you can set the poll to insuppressible. An insuppressible poll will occur for a node even if the node is suppressed. This is useful for a poll that checks the status of a node to determine if it has returned to normal.

TO ENABLE THIS ATTRIBUTE

1. From the client's **Admin** menu, choose **Poll List**.
The Poll List window is displayed.



2. Select from the list the poll whose Suppressible attribute you want to enable.
3. With your cursor positioned over the selected poll, click your right mouse button to display a pop-up menu listing actions you can take from the Poll List window.
4. Select **Suppressible** from the pop-up menu.
The poll is now suppressible, which means that the poll cannot cause NerveCenter to poll a suppressed node.

Changing Other Node Attributes

Earlier sections of this chapter explained how to change a node's property group and its Suppressed setting:

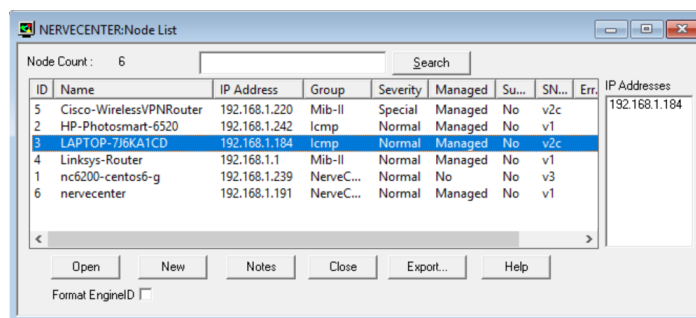
- For information on changing a node's property group, see the section [Changing a Node Property Group on page 339](#).
- For information on turning on a node's Suppressed attribute, see the section [Suppressing a Node on page 340](#).

This section explains how to change the values of a node's Managed and Auto Delete attributes.

TO CHANGE ONE OF THESE ATTRIBUTES

1. From the client's **Admin** menu, select **Node List**.

The Node List window is displayed.



2. Select a node from the list.
3. With your cursor positioned over the selected node, click your right mouse button to display a pop-up menu listing the actions you can take from this window.
4. From the pop-up menu, choose **Managed**, **Unmanaged**, **Auto Delete**, or **No Auto Delete**.

Choosing Managed is the equivalent of checking the Managed checkbox in the Node Definition window, and choosing Auto Delete is the equivalent of checking the Auto Delete checkbox. Choosing Unmanaged or No Auto Delete is the equivalent of unchecking the appropriate checkbox.

The new node setting takes effect.

Severities are NerveCenter objects that indicate the seriousness of a network or system condition. For instance, a severity is an important part of the definition of each alarm state. In the alarm definition in [Figure 60](#), you can see that the state LinkDown has the severity Major associated with it because it is colored orange.

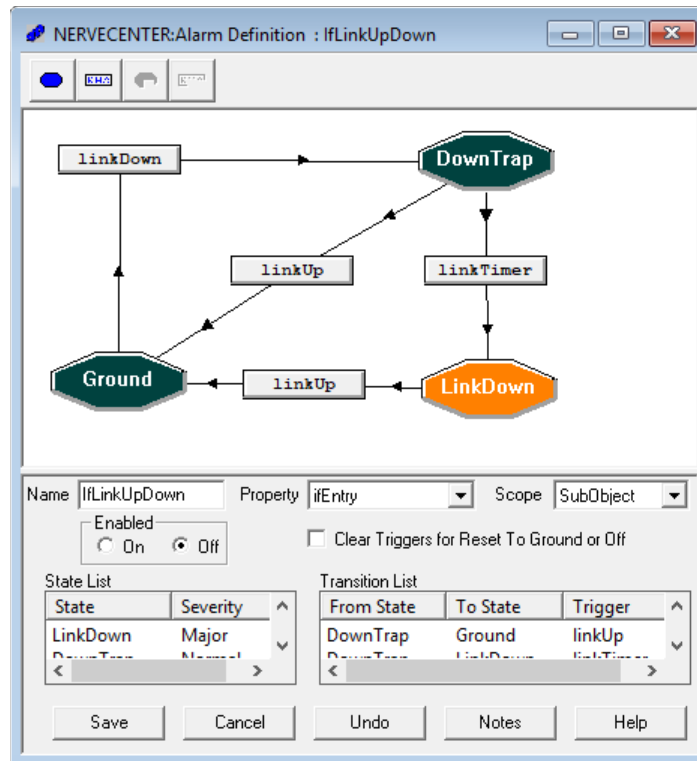


Figure 60: Alarm State Severities

In addition, NerveCenter categorizes the conditions it has detected by severity in alarm summary windows.

The remainder of this chapter explains in detail what constitutes the definition of a severity and how severities are used in NerveCenter, what predefined severities are supplied with NerveCenter, and how to create new severities.

Definition of a Severity

A severity object has the following data set described and defined in [Table 24](#).

Table 24: Definitions of Severity Attributes

Data Member	Definition
Name	A unique name.
Group	The name of the severity group to which the severity belongs. A group name should describe a general type of condition that NerveCenter can detect; for instance, the two predefined groups are Fault and Traffic, and all the predefined severities belong to one of these groups. You can also define new groups.
Color	Each severity has a color associated with it. These severity colors are used in state diagrams to indicate the severity of alarm states.
Level	A severity's level is intended to reflect the seriousness of an associated alarm state. That is, an alarm state whose severity has a level of 0 represents a harmless condition, whereas an alarm state whose severity has a high level represents a serious condition.
Platform name	The name of a severity used by your network management platform. If NerveCenter informs your platform of a condition, the platform uses the severity defined by this attribute when it displays information about the event.

For more information about these attributes, see the sections:

- [Severity Attributes Used by NerveCenter on the facing page](#)
- [Severity Attributes and Network Management Platforms on the facing page](#)

Severity Attributes Used by NerveCenter

The severity attributes Name, Group, and Color are used by NerveCenter when it displays information about current alarm instances in the Alarm Summary or Aggregate Alarm Summary window. In the figure below there is a severity named Critical, which belongs to the severity group Fault and is associated with the color red.

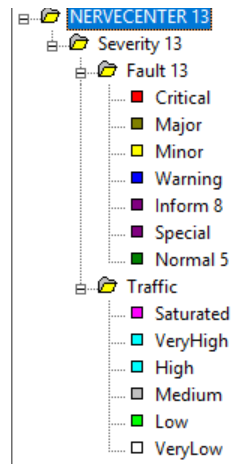


Figure 61: Severity Names, Groups, and Colors

You can add new severities to the existing groups (Fault and Traffic), or add severities that belong to a new group. In the latter case, NerveCenter will create a new folder to represent the new severity group.

Note: Severity colors are also used in alarm state diagrams to indicate the severity of particular states.

Severity Attributes and Network Management Platforms

The severity attributes Level and Platform Name are used to help define how NerveCenter interacts with a network management platform. For more details, see the NerveCenter Platform Integration Guide.

Level

Each NerveCenter severity must have a unique severity level, which is represented by an integer. You associate severities that have low severity levels with alarm states representing benign conditions, and severities that have high levels with states representing serious conditions.

Now, here's how severity levels affect NerveCenter's interaction with a network management platform. When NerveCenter is set up, an administrator can define an "Inform Configuration." This configuration indicates where NerveCenter should send messages when it performs Inform alarm actions. The configuration also specifies a "Minimum Severity." If the administrator sets the Minimum Severity to 4, only transitions to alarm states with severity levels of 4 or more can cause Inform messages to be sent to a platform.








Platform Name







You can associate with each NerveCenter severity the name of a severity defined by your network management platform. For example, the predefined severity Saturated has associated with it the platform name Normal. Given this situation, if NerveCenter sends to the platform an Inform message whose variable bindings indicate that the destination alarm state in NerveCenter had a severity of Saturated, the platform will interpret this as an event of Normal severity. That is, the event will show up in the platform's event browser as an event of Normal severity, and if the map icon representing the node whose interface was saturated is (color), that icon will remain (color).

Default Severities

Table 25 lists the thirteen predefined NerveCenter severities.

Table 25: Predefined NerveCenter Severities

Severity Name	Severity Level	Severity Group	Platform Name	Color	
Normal	0	Fault	Normal	Dark Green	
VeryLow	1	Traffic	Normal	White	
Low	2	Traffic	Normal	Yellow Green	
Medium	3	Traffic	Normal	Light Aqua	
High	4	Traffic	Normal	Cyan	
VeryHigh	5	Traffic	Normal	Sky Blue	
Saturated	6	Traffic	Normal	Magenta	

Severity Name	Severity Level	Severity Group	Platform Name	Color	
Special	7	Fault	Normal	Burgundy	
Inform	8	Fault	Normal	Violet	
Warning	9	Fault	Warning	Royal Blue	
Minor	10	Fault	Minor	Yellow	
Major	11	Fault	Major	Orange	
Critical	12	Fault	Critical	Red	

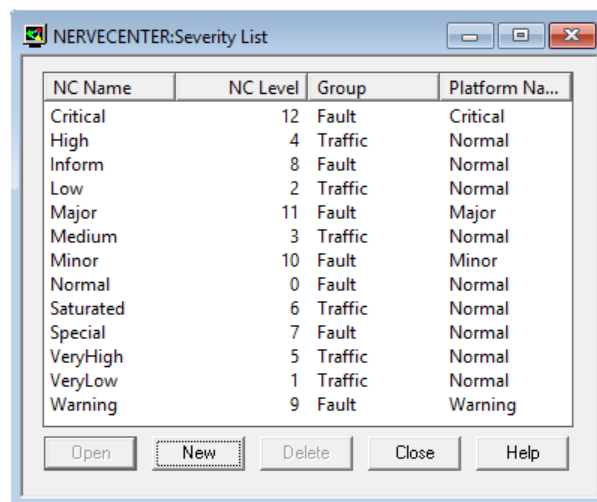
Creating a New Severity

If your behavior models require severities other than those supplied with NerveCenter, you can create new severities.

TO CREATE A NEW SEVERITY

1. From the client's **Admin** menu, choose **Severity List**.

The Severity List window is displayed. This window presents information about all the severities currently defined in the NerveCenter database.



2. Select the **New** button in the Severity List window.

The New Severity window is displayed.

New Severity

Severity Name:

Severity Level:

Severity Group:

Platform Name:

Change Color

OK Cancel Undo Help

3. Enter a unique name for your severity in the **Severity Name** field (255 characters maximum).

4. Enter a unique severity level, an integer in the range 0 to 255, in the **Severity Level** field.
Since the predefined severities use the levels 0 through 12, you should avoid those numbers (unless you've modified the levels of the predefined severities).

In general, you should set up your severity levels so that the lowest priority severities have the lowest levels and the highest priority severities have the highest levels. This is true because if NerveCenter is set up to forward information about important alarm transitions to a network management platform, NerveCenter forwards information about any transition whose destination state has a severity whose level is greater than or equal to x , where x is defined when NerveCenter is configured.

5. Enter the name of a severity group in the **Severity Group** field.
This group can be one of the preexisting groups—Fault or Traffic—or a user-defined group. In either case, the severity group should indicate the type of problem that the severity reflects.
6. In the **Platform Name** field, enter the name of a severity on your network management platform, or if you're not using a network management platform, leave the value set to "Unknown."

When you enter a platform severity name, you establish a mapping between the NerveCenter severity you're defining and a severity on your network management platform. For example, the predefined NerveCenter severity VeryHigh (traffic) is mapped by default to the platform severity Normal. Given this situation, if NerveCenter informs a platform of a condition of VeryHigh severity, the platform will indicate (in its event browser) that an event of Normal severity has occurred.

7. Assign a color to the severity.

To assign this color, perform the following steps:

- a. Select the **Change Color** button in the New Severity window.
The Color window is displayed.
 - b. Select the color box containing the color you want to assign to the severity.
 - c. Select the **OK** button in the Color window.
8. Select the **Save** button in the New Severity window.

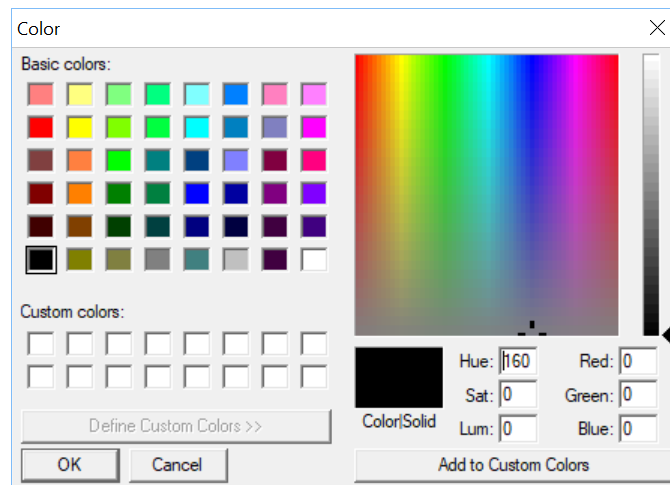
Information about the new severity is saved to the NerveCenter database.

Creating Custom Colors

One attribute of a NerveCenter severity is its color. This color can be one of 48 predefined colors or one of 16 custom (user-defined) colors. This section explains how to create a custom color that you can use later in the definition of a severity.

TO CREATE A CUSTOM COLOR

1. From the client's **Admin** menu, choose **Severity List**.
The Severity List window is displayed.
2. Select the **New** button in the Severity List window.
The New Severity window is displayed.
3. Select the **Change Color** button in the New Severity window.
The Color window is displayed. This window shows NerveCenter's predefined colors and any previously defined custom colors.
4. Select the **Define Custom Colors** button in the Color window.
The Color window expands to include an area for creating custom colors.



5. Specify the custom color you want to define by following the directions below. The color is displayed in the Color|Solid color box.
 - a. Drag the crosshairs in the large colored area horizontally to establish the desired hue.
 - b. Drag the crosshairs vertically to establish the desired amount of saturation.
Moving the crosshairs up increases the amount of saturation, and moving them down decreases the amount of saturation.
 - c. Drag the arrowhead to the right of the long, narrow colored area to establish the color's luminance.
Moving the arrowhead up increases the color's luminance, and moving it down decreases the color's luminance.

You can also specify a color by entering values in the Hue, Sat, and Lum fields or the Red, Green, and Blue fields.
 6. Select the color square in the "Custom color" area to in which you want to save the new custom color.
You can overwrite an existing custom color with a new one.
 7. Select the **Add to Custom Colors** button.
The new color is saved and is available for assignment to a severity.
-

Importing and Exporting NerveCenter Nodes and Objects

The NerveCenter Client import and export features enable you to choose which NerveCenter behavior models, objects, or nodes to import or export. Perhaps you have developed a behavior model that you want to propagate across a multi-NerveCenter server environment. With the export feature, you can selectively load one or more behavior models, (or individual objects) into another NerveCenter server's database.

In addition to directly exporting to another NerveCenter server's database, you can also export NerveCenter objects, nodes, and behavior models to a file. Using the import feature, you then import such files into a NerveCenter database. For example, you might want to create a master node list and then divide it into smaller lists to export to remote NerveCenter installations. Or, perhaps, create a node list as a backup for quick recovery should the system go down.

For a complete list of the types of NerveCenter objects that you can export, see the section, [Exporting NerveCenterObjects and Nodes to Other Servers on page 357](#).

NerveCenter ships with object and behavior model files (.mod) that include fixes and vendor-specific behavior models. Because not everyone will want to use them, these objects and models are not loaded into the NerveCenter database by default. With the import feature, you can load these definitions into your NerveCenter database.

Exporting Behavior Models to Other Servers

When you don't want to export an entire NerveCenter database, NerveCenter enables you to pick and choose those behavior models you want to export to other NerveCenter servers. For example, for a multi-NerveCenter site, you might want to propagate particular behavior models across your NerveCenter servers.

To export behavior models to a file, see [Exporting Behavior Models to Other Servers above](#). For information about exporting a set of nodes or individual NerveCenter objects, see the following sections:

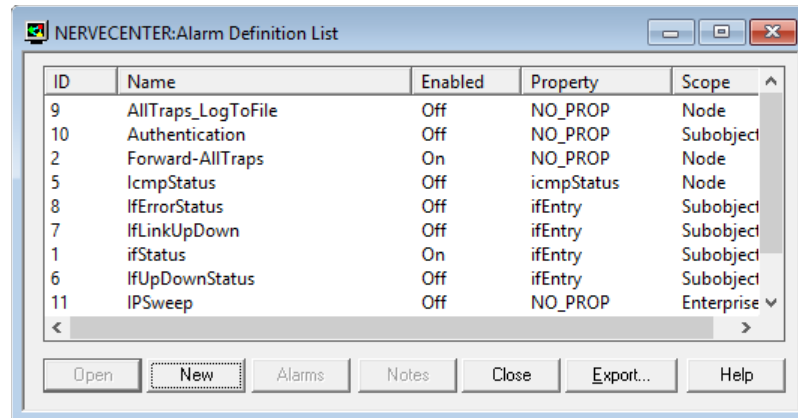
- [Exporting NerveCenterObjects and Nodes to Other Servers on page 357](#)
- [Exporting NerveCenterObjects and Nodes to Other Servers on page 357](#)

TO EXPORT BEHAVIOR MODELS TO ANOTHER NERVECENTER SERVER

1. Be sure that you are connected to the NerveCenter server(s) to which you want to export the behavior model. (See [Connecting to a Server on page 54](#) for more information.)

2. From the client's **Admin** menu, choose **Alarm Definition List**.

The Alarm Definition List window is displayed.

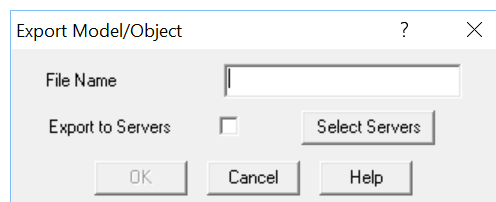


3. Select the alarm whose behavior model you want to export.

You can select any number of alarms at one time.

4. Right-click the selected alarm to bring up the alarm pop-up menu, and select **Export Model**.

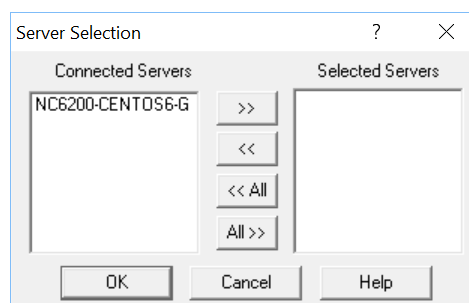
The Export Model/Object dialog is displayed.



5. Select the **Export to Servers** checkbox.

6. Select the **Select Servers** button.

The Server Selection dialog box is displayed.



- a. Select the servers to which you're exporting from the list.
 - b. Select the >> button. To select all servers to export to, select the **All >>** button.
The selected servers are added to the Selected Servers list.
You can remove servers from the Selected Servers list by selecting the object and then selecting the << button.
Repeat this step for each server to which you want to export behavior models.
 - c. When finished, select **OK** to save your choices and close the Server Selection dialog.
7. Select the **OK** button
The behavior model(s) you've selected are exported to the selected NerveCenter server(s)' database.
-

Exporting Behavior Models to a File

Situations can arise when you might want to export particular NerveCenter behavior model to a file. Having one or more behavior models in a separate file can be useful when troubleshooting NerveCenter or sharing behavior models between different NerveCenter sites.

For more about what NerveCenter actually exports when you select an behavior model, see the section [Exporting Behavior Models to Other Servers on page 353](#).

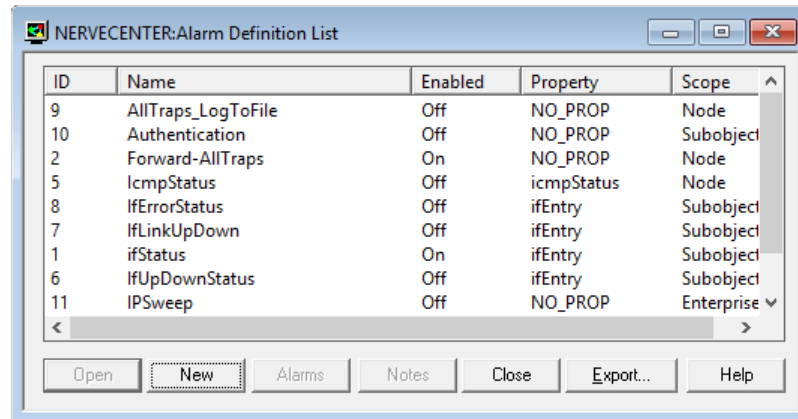
When you export one or more behavior models to a file, NerveCenter actually creates two files:

- A file with a .mod extension that contains the data required to re-create the behavior models. This is the file that is imported later into the destination database.
- A text file (*.txt) that contains a textual description of the exported behavior models. Although not required during an import, this file is important because it serves as documentation for the corresponding .mod file and is the only method of knowing what models reside in the .mod file prior to actually importing the models.

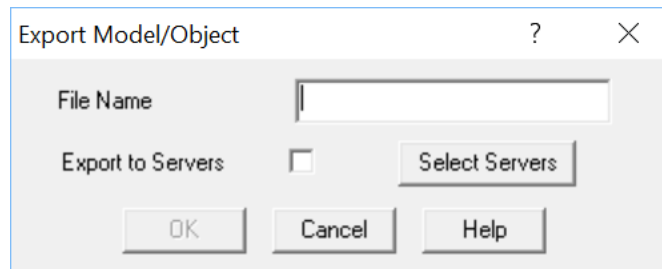
TO EXPORT BEHAVIOR MODELS TO A FILE

1. From the client's **Admin** menu, choose **Alarm Definition List**.

The Alarm Definition List window is displayed.



2. Select the alarm whose behavior model you want to export.
You can select any number of alarms at one time.
3. Right-click the selected alarm to bring up the alarm pop-up menu, and select **Export Model**.
The Export Model/Object dialog is displayed.



4. In the **File Name** text field, type a filename without an extension or a pathname including a filename without an extension.

NerveCenter will create two files. One will have the filename extension `.mod` and contain the actual data for the behavior model you export. This is the file that you can import into another NerveCenter database. The second file will have a `.txt` extension and contain a textual description of the behavior model. This file is not used during an import operation, but it is the only source of documentation for the `.mod` file contents.

If you specify a pathname in the File Name field, the file will be written to the directory you specify. By default, NerveCenter places the file in the NerveCenter model directory.

5. Select the **OK** button.

More About Exporting Behavior Models

When you export a behavior model to another NerveCenter server or to a file, you export an alarm (or alarms) and all of the objects associated with that alarm. These associated objects include:

- Any object that can fire a trigger that can cause a transition in the alarm, including polls, masks, and other alarms.
- All triggers referred to in any exported object, including:
 - For masks, simple trigger and triggers used in the trigger function.
 - For alarms, triggers used in Clear/Fire Trigger actions.
 - For polls, True and False trigger.
- Any alarm that can be affected by a trigger fired by the alarm.
- Any properties used by any of the exported objects.
- Any property groups that contain any of the properties mentioned above.

Note: The exported property groups include only those properties used by the behavior model.

- Any property groups used in AssignPropertyGroup() functions in polls, masks, and Perl Subroutine expressions. Also, any property groups used in SetAttribute alarm actions in alarm transitions. No properties are included from the group.
- The severities used by the exported alarms.
- Any Perl subroutines called by a Perl Subroutine action in an exported alarm.

NerveCenter does *not* export the following objects with behavior models:

- Alarms that listen to Clear Trigger alarm actions.
- Objects that fire triggers used only in Clear Trigger alarm actions of the exported alarms.
- Polls and trap masks that fire triggers used only in Fire Trigger alarm actions of the exported alarms. (Perl subroutines in this situation *are* exported.)
- Perl subroutines that are not used as an action in one of the exported alarms.
- Action Router rules.

Exporting NerveCenterObjects and Nodes to Other Servers

When you don't want to export an entire NerveCenter database, NerveCenter enables you to pick and choose those nodes and objects you want to export to other NerveCenter servers. For example, for a multi-NerveCenter site, you might want to propagate particular masks across your NerveCenter servers.

Caution: If you export nodes to a NerveCenter Server on another segment, any applicable parenting information is exported with the nodes. However, this information might not be valid for the new topology into which the node information is imported.

For a complete list of the object types and what NerveCenter actually exports when you select an object, see the section [Exporting NerveCenter Objects and Nodes to Other Servers](#) on the previous page.

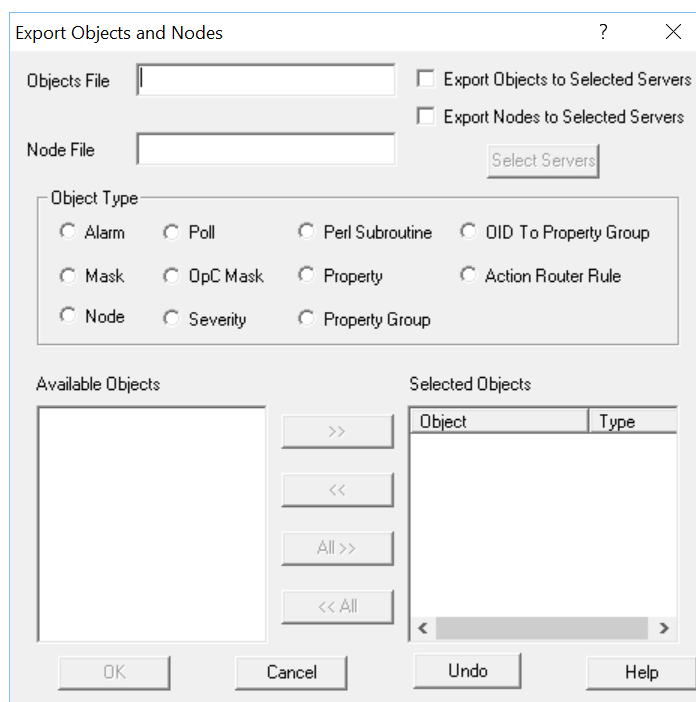
To export nodes and objects to a file, see "Exporting NerveCenter Objects and Nodes to a File" on [page 360](#). For information about exporting a behavior model—an alarm and all of the objects associated with it—see the following sections:

- [Exporting Behavior Models to Other Servers on page 353](#)
- [Exporting Behavior Models to a File on page 355](#)

TO EXPORT A SET OF NODES OR OBJECTS TO ANOTHER NERVECENTER SERVER

1. Be sure that you are connected to the NerveCenter server(s) to which you want to export the nodes or objects. (See [Connecting to a Server on page 54](#) for more information.)
2. From the client's **Admin** menu, choose **Export Objects and Nodes**.

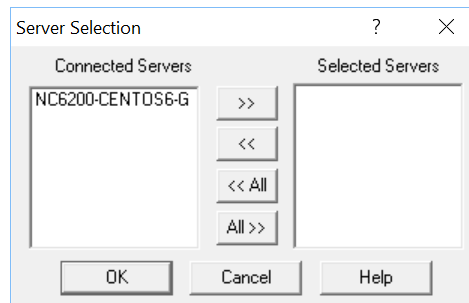
The Export Objects and Nodes dialog is displayed.



3. To export:
 - **Objects**—select the **Export Objects to Selected Servers** to choose servers for objects you're exporting.
 - **Nodes**—select **Export Nodes to Selected Servers** to choose servers for nodes you're exporting.

Caution: If you export nodes to a NerveCenter Server on another segment, any applicable parenting information is exported with the nodes. However, this information might not be valid for the new topology into which the node information is imported.

4. Select the **Select Servers** button.
The Server Selection dialog is displayed.



- a. Select the servers to which you're exporting from the list.
 - b. Select the **>>** button. To select all servers to export to, select the **All >>** button.
The selected servers are added to the Selected Servers list.
You can remove servers from the Selected Servers list by selecting the object and then selecting the **<<** button.
Repeat this step for each server to which you want to export objects or nodes.
 - c. When finished, select **OK** to save your choices and close the Server Selection dialog.
5. Select Node or the type of object that you want to export from the **Object Type** radio set.
 6. Create a list of nodes or objects to be exported. Creating a node or object list is similar to how you selected the server(s) in [Step 4](#).
The selected objects or nodes are added to the Selected Objects list.
Repeat [Step 5](#) and [Step 6](#) for each type of object that you want to export.
7. Select the **OK** button.
The definition of the objects or nodes you've selected are exported to the selected NerveCenter server(s)' database.

Exporting NerveCenterObjects and Nodes to a File

Situations can arise when you might want to export particular NerveCenter nodes and objects to a file. Having nodes or objects in a separate file can be useful when troubleshooting NerveCenter or sharing nodes and objects between different NerveCenter sites.

For a complete list of the object types and what NerveCenter actually exports when you select an object, see the section [Exporting NerveCenterObjects and Nodes to Other Servers on page 357](#).

When you export *objects* to a file, NerveCenter actually creates two files:

- A file with a .mod extension that contains the data required to re-create the objects. This is the file that is imported later into the destination database.
- A text file (*.txt) that contains a textual description of the exported objects. Although not required during an import, this file is important because it serves as documentation for the corresponding .mod file and is the only method of knowing what models reside in the .mod file prior to actually importing the models.

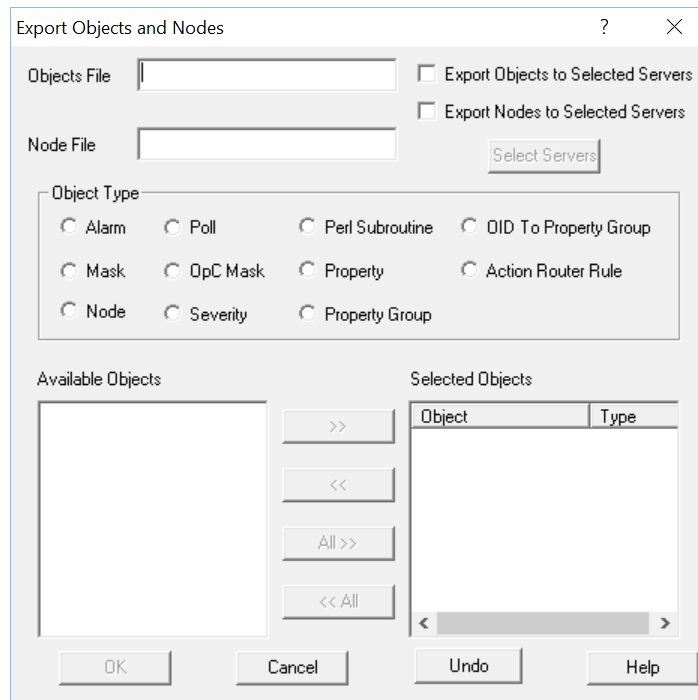
When you export *nodes*, NerveCenter creates a .node file that contains the data to re-create the nodes.

To export nodes and objects to a NerveCenter database on another NerveCenter server, see [Exporting NerveCenterObjects and Nodes to Other Servers on page 357](#). For information about exporting a behavior model—an alarm and all of the objects associated with it—see the following sections:

- [Exporting Behavior Models to Other Servers on page 353](#)
- [Exporting Behavior Models to Other Servers on page 353](#)

TO EXPORT A SET OF OBJECTS FROM NERVECENTER

1. From the client's **Admin** menu, choose **Export Objects and Nodes**.
The Export Objects and Nodes dialog is displayed.



2. In the Objects File text field, type a filename for the serialized text file you want to export. You can include the path in order to write the file to a certain location; by default, NerveCenter places the file in the NerveCenter installation/model directory.
NerveCenter will create two files. One will have the filename extension .mod and contain the actual data for the objects you export. This is the file that you can import into another NerveCenter database. The second file will have a .txt extension and contain a textual description of the objects. This file is not used during an import operation, but it is the only source of documentation for the .mod file contents.
3. If you are exporting nodes, NerveCenter also creates a .node file by default in the model directory. You must provide a name for this file in the Node File field. This file can later be imported using the importutil.exe tool, which is described in [Reconfiguring a NerveCenter Server from the Command Line](#) in Managing NerveCenter.

4. Create a list of objects to be exported by following the directions below:
 - a. Select the radio button for the type of object you want to export, such as Property Group. A list of objects of that type is displayed in the Available Objects list box.
 - b. Select the objects you want to export from the list.
 - c. Select the >> button. To select all objects for export, select the **All >>** button. The selected objects are added to the Selected Objects list; you can remove objects from the Selected Objects list by selecting the object and then selecting the << button. Repeat this step for each type of object that you want to export.
 5. Select the **OK** button. The definition of the objects you've selected are exported.
-

More about Exporting Objects

Using the client's Export Objects and Nodes command (Admin menu), you can export the following NerveCenter objects:

- "Alarm" on the facing page
- "Mask" on the facing page
- "OID to Group" on the facing page
- "Perl Subroutine" on the facing page
- "Poll" on the facing page
- "Property" on the facing page
- "Property Group" on the facing page
- "Rule" on the facing page
- "Severity" on page 364

When you export an object to another server, NerveCenter actually exports not only that object, but any objects that the object contains and some related objects. [Table 26](#) lists the objects that NerveCenter exports for each object type.

Table 26: Exporting Objects

Object Type	Objects Exported
Alarm	<ul style="list-style-type: none"> ■ The alarm ■ The alarm's property ■ Any property groups that contain the alarm's property or is used by the set attribute action ■ The triggers that can affect the alarm, including those called by Perl subroutines ■ The severities used by the alarm's states ■ Any action, including but not limited to Perl subroutines called by a Perl Subroutine action
Mask	<ul style="list-style-type: none"> ■ The mask ■ The triggers fired by the mask, including simple triggers and triggers referenced by FireTrigger() ■ Any property groups referenced by the trigger function AssignPropertyGroup()
OID to Group	<ul style="list-style-type: none"> ■ The OID to property group mapping ■ The property group referred to in the mapping
Perl Subroutine	<ul style="list-style-type: none"> ■ The Perl subroutine ■ Any property groups referenced by AssignPropertyGroup() ■ Any triggers referenced by FireTrigger()
Poll	<ul style="list-style-type: none"> ■ The poll ■ The poll's property ■ Any property groups that contain the poll's property, including any referenced by AssignPropertyGroup() ■ The triggers fired by the poll, including any referenced by FireTrigger()
Property	<ul style="list-style-type: none"> ■ The property ■ The property groups that contain the property
Property Group	<ul style="list-style-type: none"> ■ The property group ■ The properties in the property group ■ Any property groups that are a superset of this property group
Rule	<ul style="list-style-type: none"> ■ The Action Router rule conditions and actions

Object Type	Objects Exported
Severity	<ul style="list-style-type: none"> ■ The severity name ■ The severity color ■ The severity numeric level ■ The severity mapped name, if applicable

Importing Node, Object, and Behavior Model Files

You can copy node, object, and behavior model definitions into another NerveCenter database. Node files (.node) contain node definitions that have been exported to a file with the NerveCenter export feature. Object and behavior model files (.mod) contain definitions of objects and behavior models. Object/model files come from one of two places:

- Object or behavior model files created using NerveCenter's export feature.
- Models files shipped with NerveCenter. These files reside in NerveCenter's model directory.

When you import a behavior model, you are also importing the objects associated with that model. For every object/model file (.mod) there is a text file that contains descriptions of the objects in the model. (This text file is the *only* documentation for the .mod file.)

Caution: Any existing object with the same name as an imported object is overwritten.

Whatever the source of your node or object/model files, and regardless of whether they contain individual objects or behavior models, you use the same procedure to import the file contents.

Note: You can also use the utility ImportUtil to import behavior models. This utility is discussed in [Importing Behavior Models or Nodes with ImportUtil on the facing page](#).

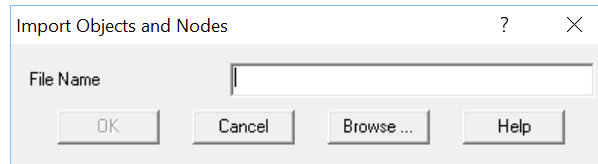
TO IMPORT THE CONTENTS OF A NODE OR OBJECT/MODEL FILE

1. If the objects you are importing use base objects or attributes not in the current NerveCenter MIB, add the necessary MIB definitions and recompile the NerveCenter MIB before proceeding. See [Managing Management Information Bases \(MIBs\)](#) in *Managing NerveCenter*.

Note: IP filters set in the NerveCenter Administrator also apply to nodes imported via a node file. For more information, refer to [Filtering Nodes](#) and [Enabling and Disabling IP and Hostname Filters](#) in *Managing NerveCenter*.

2. Move the node or object/model file to a location available to the destination NerveCenter server.

- From the client's Server menu, choose **Import Objects and Nodes**.
The Import Objects and Nodes dialog is displayed.



- In the **File Name** field, enter the path of the node or object/model file.

Caution: Any existing object with the same name as an imported object is overwritten.

If you don't specify a pathname, NerveCenter looks in the server's current working directory; the server's current working directory is always the NerveCenter bin directory.

- Select the **OK** button in the Import window.
NerveCenter imports the node or object/model file definitions into the new server's database.

Note: If you are missing objects in a behavior model you have imported, you must update the compiled NerveCenter MIB file. Adding MIB definitions is described in [Managing Management Information Bases \(MIBs\)](#) in *Managing NerveCenter*. (For any models that you imported *before* you updated and recompiled the NerveCenter MIB, the missing objects will not appear until the alarms they transition are instantiated or until you re-import the model/objects.)

Importing Behavior Models or Nodes with ImportUtil

You can use the utility `importutil` to copy server configuration information into the registry of a new NerveCenter Server and to import behavior models and nodes. Using this utility allows you to reconfigure a setting on more than one NerveCenter Server at a time by changing one file and importing it to all the relevant servers.

For more details, see [Reconfiguring a NerveCenter Server from the Command Line](#).

TO IMPORT USING IMPORTUTIL

1. Locate the file **imputil.ini**.
 - In a typical NerveCenter installation on UNIX, this file can be found in the directory `/opt/OSInc/userfiles`.

The file `imputil.ini` is made of a number of sections that include a section header and keys.

2. Before making any changes, create a backup copy of the file `imputil.ini`.

Caution: You will not be able to restore the original `imputil.ini` after making changes to the file unless you first make a backup copy.

3. Delete all but the `[IMPORT_MODEL]` or `[IMPORT_NODE]` section, as appropriate.
All sections in the file are optional. If you remove a section, including the section header, `ImportUtil` does not change or delete any values in the registry for that key.

Any new values left in `imputil.ini` will overwrite old values. To avoid having placeholders overwrite legitimate values, delete any unnecessary keys before running `ImportUtil`.

4. Enter the file name and path of the model or node list you want to import.

```
[IMPORT_MODEL]
File = model_path_and_filename
```

or

```
[IMPORT_NODE]
File = node_path_and_filename
```

Remember that you must also add the MIB definitions to the NerveCenter MIB file. Refer to [Managing Management Information Bases \(MIBs\)](#) in *Managing NerveCenter* for information about changing the NerveCenter MIB file.

5. Save the changed file.
6. While the NerveCenter Server is running, run the utility `ImportUtil` using the following command:

```
importutil imputil.ini
```

Note: You must either be in the same directory as the `imputil.ini` file or include the full pathname of the `imputil.ini` file.

NerveCenter notifies you upon successful completion of the reconfiguration.

If NerveCenter is running as a service, the account under which the service or daemon is running must have the appropriate rights/group membership to import from or export to local machine or network shares.

As a tool that comprehensively monitors and manages your network, NerveCenter uses a variety of data transfers to gather, correlate, disseminate, and store information about network events. This appendix outlines the general flow of data into, through, and out of NerveCenter in the course of its operation.

NerveCenter's primary sources of network information are SNMP traps and device responses to NerveCenter polls. If configured appropriately, LogMatrix NerveCenter responds to trap and poll data by forwarding it to your network management platform and to other NerveCenters. For example, forwarded event data might ultimately land in a network management platform's Event Categories window or trigger an alarm transition in a central NerveCenter. Although this sequence may happen quickly, the actual communication path from initial receipt of trap or poll data to the final event message has many stages.

As [Figure 62](#) shows, a trace of the communication path initiated by a managed device's SNMP trap or poll response might look like this:

1. Traps are relayed directly to the NerveCenter Server if the platform and the server are running on different machines. If they're running on the same machine, traps are detected by the operating system trap service or the management platform's trap service and then forwarded to the NerveCenter SNMP Trap process. The NerveCenter SNMP Trap process, in turn, forwards the trap to LogMatrix NerveCenter.
2. LogMatrix NerveCenter *trap masks* filter incoming traps to see if they are of interest. If a trap is of interest, an internal event, called a *trigger*, is generated and used by active *alarms*. Polls evaluate the poll data returned by managed devices and also use triggers to pass data to alarms.
3. LogMatrix NerveCenter alarms correlate the traps and polls with other related data. For example, an alarm might detect that this is the third trap of the same type from the same machine. The alarm then takes any automated actions that were associated with this trap detection. For example, it could issue a trouble ticket or change the device configuration.

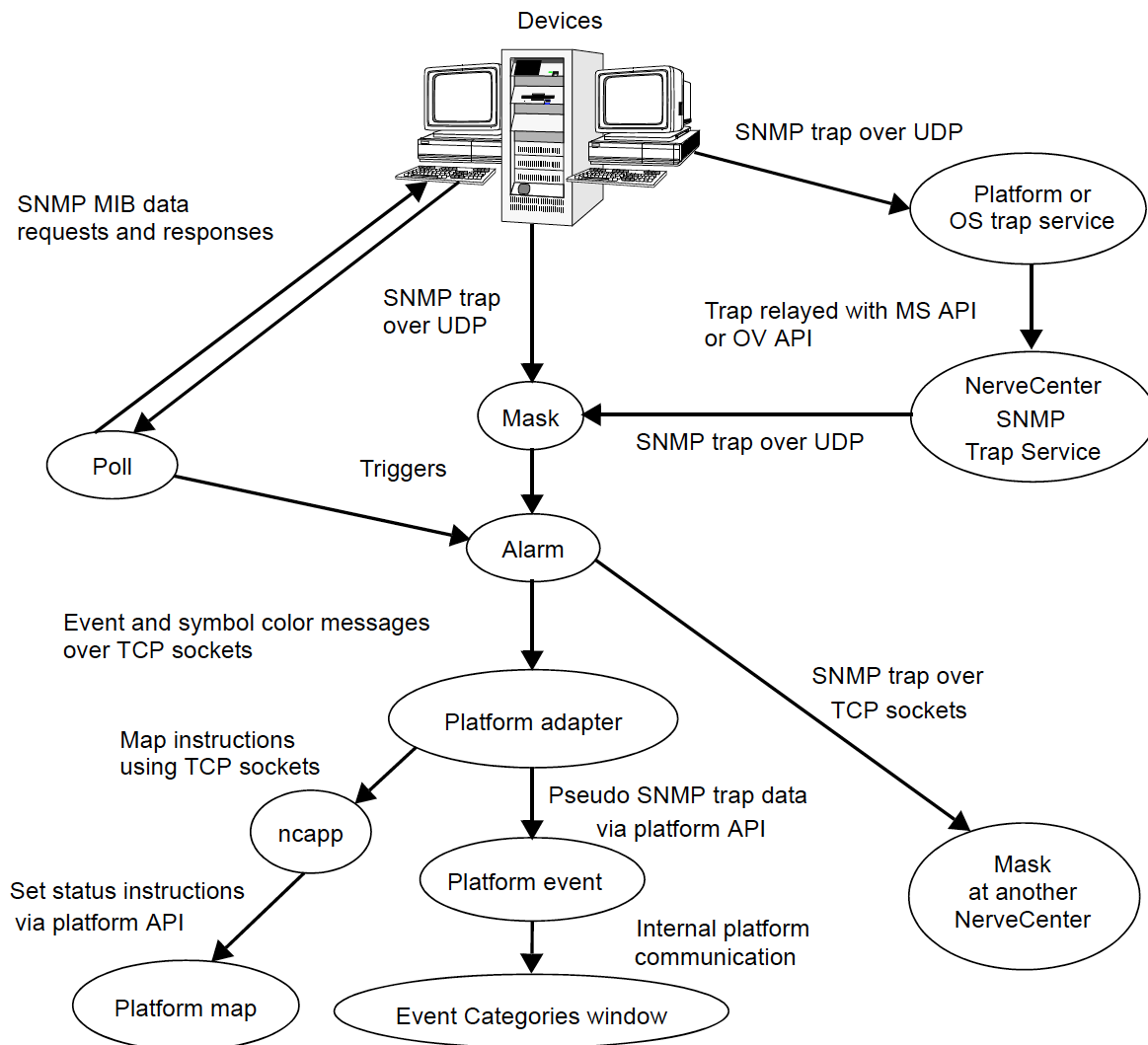


Figure 62: Data Flow

4. If an alarm transition contains the Inform action, the alarm sends a message to the LogMatrix NerveCenter platform adapter process, which always resides on the same host as the network management platform, and/or to any listed NerveCenters.
5. The platform adapter determines whether the message requires changing a symbol's color on the map, initiating an event message, or both. Messages to other NerveCenters forward the trap data.
6. If an event is to be posted, the platform adapter uses an API to submit a data structure that resembles an SNMP trap to the platform event facility, which decodes traps, associates text messages with events, and posts them in the Event Categories window.

NerveCenter is a client/server application. The NerveCenter server acts as the hub for the data transfers described in this appendix. As shown in the following illustration, event information moves from managed device to NerveCenter server to management platform. But data also flows between the server and other NerveCenter components in support of this flow.

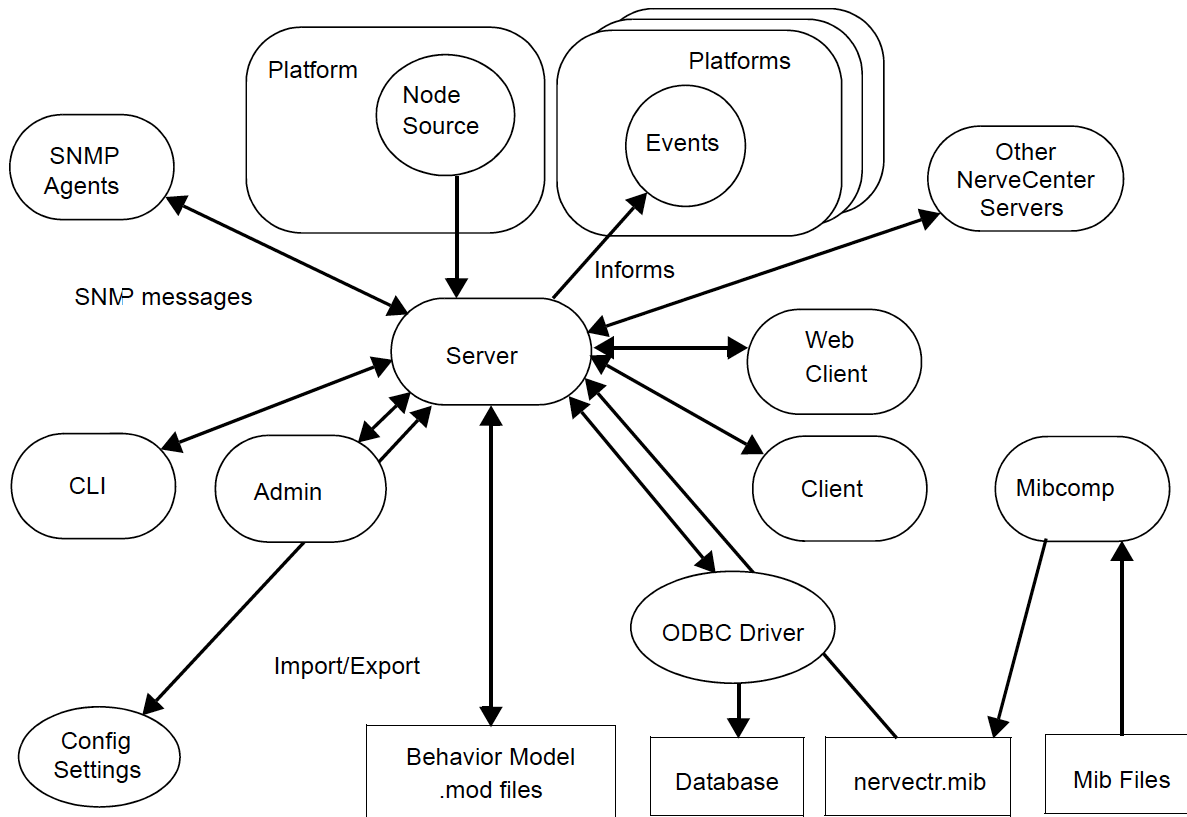


Figure 63: NerveCenter Components

The components shown in the preceding figure are defined in [Table 27](#):

Table 27: NerveCenter Components

Component	Definition
Client	A user interface to the server. Provides facilities for the creation, modification, maintenance, and monitoring of behavior models.
Administrator	A user interface to the server. Provides facilities for NerveCenter configuration.
Command line interface (CLI)	Provides a subset of client commands for use from the command line, programs, and scripts.
Platform/node source	The network management platform that provides and monitors a list of nodes to be monitored by the server.
Helps you plan and carry out your NerveCenter upgrades and new installations. Use the <i>Release Notes</i> in conjunction with this book.	The network management platforms that the server informs as an alarm action.
Other NerveCenters	Other NerveCenter servers that can accept Informs from the server, allowing correlation across multiple domains.
SNMP agents	Agents running on managed nodes that generate traps and respond to NerveCenter polls.
mibTool	Utility to compile and merge MIBs into the NerveCenter master MIB.
Configuration Settings	Repository for NerveCenter configuration parameter values— NerveCenter.xml configuration file (UNIX) and the Registry (Windows).
Behavior model .mod files	ASCII files containing exported behavior models and their components.

This appendix provides information for resolving problems relating to NerveCenter behavior models. Actions you can take to debug behavior models include:

- Verifying that the behavior model is enabled
- Checking properties and property groups
- Matching triggers and alarm transitions
- Auditing behavior models

Enabling a Behavior Model's Components

If a behavior model is not working, the first thing to check is whether all of the model's components have been enabled. For a model to be functional, all polls, masks, and alarms must be enabled.

To determine whether a given object is enabled, open the Poll List, Mask List, or Alarm Definition List window, and note the Enabled status of the object in which you're interested. For information of how to enable an object, see the section [Enabling Objects on page 333](#).

Checking Properties and Property Groups

If all of the components of a behavior model are enabled and the behavior model still does not work, you should make sure that your polls' properties, your alarms' properties, and your nodes' property groups are set up correctly. The upcoming sections explain how to perform these checks.

Checking a Poll's Property

Part of NerveCenter's smart polling feature is that NerveCenter does not send a poll to a node unless the poll's property is in the node's property group.

TO MAKE SURE THAT YOUR POLL PASSES THIS TEST

1. Open the Poll List window, and note your poll's property.
If your poll's property is set to NO_PROP, you can stop the test here because a poll whose property is NO_PROP always passes this test.
2. Open the Node List window, locate a node you are trying to poll, and note this node's property group.
3. Open the Property Group List window, select the property group you noted in [Step 2](#), and see whether the poll's property appears in the property group's list of properties.
If your poll's property is not in the node's property group, you must change your poll's property, change the node's property group, or add a property to the current property group.

Checking a Poll's Poll Condition

If your poll's poll condition refers to a MIB base object, NerveCenter does not send the poll to a node unless the base object referred to in the poll condition is in the node's property group.

TO MAKE SURE THAT YOUR POLL PASSES THIS TEST

1. Open the Poll List window, and note your poll's base object.
If your poll's base object is set to NO_OBJECT, you can stop the test here because a poll whose base object is NO_OBJECT always passes this test.
2. Open the Node List window, locate a node you are trying to poll, and note this node's property group.
3. Open the Property Group List window, select the property group you noted in [Step 2](#), and see whether the poll's base object appears in the property group's list of properties.
If your poll's base object is not in the node's property group, you must change the node's property group or add a property to the current property group.

Checking an Alarm's Property

Let's assume that NerveCenter is polling a node, that NerveCenter is firing a trigger as a result of the poll, and that you have an enabled alarm whose one transition out of the Ground state has the same name as this trigger. Even in this case, NerveCenter does not create an alarm instance unless the alarm's property is in the node's property group.

TO MAKE SURE THAT YOUR ALARM PASSES THIS TEST

1. Open the Alarm Definition List window, and note your alarm's property.
If your alarm's property is set to NO_PROP, you can stop the test here because an alarm whose property is NO_PROP always passes this test.
2. Open the Node List window, note the property group of a node you are trying to poll.
3. Open the Property Group List window, select the property group you noted in [Step 2](#), and see whether the alarm's property appears in the property group's list of properties.
If your alarm's property is not in the node's property group, you must change your alarm's property, change the node's property group, or add a property to the current property group.

Matching Triggers and Alarm Transitions

When a trigger is fired, NerveCenter must decide whether that trigger should cause a state transition in an active alarm instance or cause a new alarm instance to be created. What conditions must a trigger and transition meet before one of these actions takes place?

- A transition whose name matches the name of the trigger must be pending.
In an active alarm, a transition is pending if its origin state is the alarm instance's current state. A transition is also considered pending if its origin state is Ground. When the second type of transition occurs, a new alarm instance is instantiated.
- The trigger's identity must match the transition's identity.
Triggers have four-part identities. These identities include a name, a subobject, a node, and sometimes a property. Transitions' identities have the same four parts, plus a fifth part, scope. NerveCenter uses matching rules to compare a trigger's identity to the identity of each pending alarm transition. Each pair of names, subobjects, nodes, and properties must pass a comparison test before a transition takes place.

This section describes the identities of triggers and transitions, specifies the matching rules, and provides examples of objects that match and objects that don't match. See the subsections listed below:

- [Enabling a Behavior Model's Components on page 371](#)
- [Enabling a Behavior Model's Components on page 371](#)
- [Enabling a Behavior Model's Components on page 371](#)

Identities of Triggers and Transitions

The components of a trigger's identity may be supplied by you, the designer, or by NerveCenter, depending on how the trigger is generated. On the other hand, a transition's identity is inherited from an active alarm instance or, if the transition's origin state is Ground, from an alarm definition. The remainder of this section discusses how the components of a trigger or transition's identity are given values.

- **Name**—Any string.
 - **Trigger**—You give a trigger its name when you define the poll or mask that will fire the trigger, when you make a call to the `FireTrigger()` function, or when you use the Fire Trigger alarm action. NerveCenter assigns reserved names to built-in triggers.
 - **Transition**—You establish a transition's name when you define the transition, in the course of drawing an alarm's state diagram.
- **Subobject**—Usually the MIB base object and instance (connected with a period) associated with the condition that prompted the trigger.
 - **Trigger** —The subobject of a trigger fired by a poll is taken from the OID used in the SNMP GetRequest that caused the trigger to be fired. Similarly, the subobject of a trigger fired by a trap mask is taken from the OID in the first variable binding in the trap that caused the trigger to be fired. Built-in triggers are assigned a subobject of \$ANY.

For triggers fired as a result of a call to the Fire Trigger () function or by a Fire Trigger alarm action, you specify the subobject when you call the function or define the alarm action.
 - **Transition** — In a subobject-scope alarm instance, a transition inherits its subobject from the alarm instance. For example, if an alarm instance tracks `ifEntry.2` on a given node, all its transitions do also. If the transition would be an alarm instance's first, it has no subobject. Transitions in node- and enterprise-scope alarms do not have subobjects either.
- **Node** — The name of a managed node.
 - The node attribute of a trigger fired by a poll or a mask is assigned the name of the node on which the condition of interest was detected. For triggers fired as a result of a call to the Fire Trigger () function or by a Fire Trigger alarm action, you specify the node when you call the function or define the alarm action.
 - A transition inherits its node from its alarm instance. For example, if an alarm instance tracks node `router1`, all of its transitions do also. If the transition would be an alarm instance's first, the transition does not have a node. In addition, transitions in enterprise scope alarms do not have nodes.
- **Property** — The name of a property or empty.
 - **Trigger** — You specify the property of a trigger fired by a Fire Trigger alarm action when you define the action. Triggers from other sources do not have properties.
 - **Transition** — A transition inherits its property from the associated alarm definition.
- **Scope** — Subobject, Node, Instance, or Enterprise
 - **Trigger** — A trigger does not have a scope.
 - **Transition** — A transition inherits its scope from the associated alarm definition.

Rules for Matching

A trigger causes an alarm transition if the identities of the trigger and the transition match—that is, if their names, subobjects, nodes, and properties all pass comparison tests. The four comparison tests corresponding to the four parts of a trigger's identity are discussed in the upcoming subsections. The trigger must pass all four tests before it can prompt a transition.

Name Rule

A trigger's name must match the transition's name exactly.

Subobject Rule

A trigger's subobject matches a transition's subobject when *any* of the following statements is true:

- The transition's scope is Enterprise.
- The transition's scope is Node.
- Both the trigger's and the transition's subobjects are zero instance (*baseObject.0*) or are empty.
- The trigger's subobject matches the transition's subobject exactly.
- The transition's scope is instance and the instances match.
- The trigger's subobject is a wildcard (\$ANY), and the transition's origin state is not Ground.
- The transition has subobject scope, the base objects are the same in the subobject for the trigger and transition, the instance in the trigger's subobject is a wildcard (\$ON), and the transition's origin state is not Ground.
- The transition has instance scope, the instance in the trigger's subobject is a wildcard (\$ON), and the transition is not from ground state.
- The instances in the trigger's subobject and transition's subobject match, and one of the base objects is an extension of the other.

Here's an example of one base object extending another. MIB-II defines *ifEntry*, a row of data in a table of information about an interface. You access a particular instance of *ifEntry* using the index *ifIndex*. Cisco extends this interface table by defining a local interface table, which contains many additional attributes for each interface. The rows in this table are accessed using the same index used to access the rows in the MIB-II interface table.

If the transition's origin state is Ground -- that is, a new alarm instance is being created -- the following statement must also be true:

- The trigger's subobject is not \$ANY or \$NULL and does not contain \$ON.

The trigger can have an empty subobject.

Node Rule

A trigger's node matches a transition's node when *any* of the following statements is true:

- The transition's scope is Enterprise.
- The trigger's node matches the transition's node exactly.
- The trigger's node is \$ANY, and the transition's origin state is not Ground.

If the transition would create a new alarm instance and therefore has no associated node, the following statement must also be true:

- The trigger's node is not \$ANY.

Property Rule

A trigger and transition pass the property test when *all* of the following conditions are met:

- For transitions of subobject or node scope, the transition's property is contained in the property group of the trigger's node, or the transition's property is NO_PROP.
- For transitions of subobject or node scope, the trigger's property (if it has one) is contained in the property group assigned to the trigger's node.
- For transitions of enterprise scope, the trigger's property (if it has one) must match the transition's property.

Examples of Matching Triggers and Transitions

This section presents a number of examples of triggers and transitions that do and do not match.

Example 1

A trigger named highLoad with the subobject system.0 and the node hp124 *would* prompt the following transitions:

- **Name:** highLoad
Scope: Subobject
Subobject: ip.0
Node: hp124
Property: hpws, which is contained in hp124's property group
- **Name:** highLoad
Scope: Subobject
Subobject: Unassigned (transition from Ground)
Node: Unassigned (transition from Ground)
Property: NO_PROP
- **Name:** highLoad
Scope: Node
Subobject: Irrelevant
Node: hp124
Property: hpws, which is contained in hp124's property group

The highLoad trigger *would not* prompt the following transition:

- **Name:** highLoad
Scope: Subobject
Subobject: ifEntry.2
Node: hp124
Property: hpws, which is contained in hp124's property group

The trigger and transition fail the subobject rule.

Example 2

A trigger named lowSpace with the subobject \$ANY, the node hp124, and the property includeMe (which is contained in hp124's property group) *would* prompt the following transitions:

- **Name:** lowSpace
Scope: Subobject
Subobject: ifEntry.2
Node: hp124
Property: includeMeToo, which is contained in hp124's property group
- **Name:** lowSpace
Scope: Node
Subobject: Irrelevant
Node: hp124
Property: NO_PROP
- **Name:** lowSpace
Scope: Subobject
Subobject: system.0
Node: hp124
Property: NO_PROP

The lowSpace trigger *would not* prompt the following transitions:

- **Name:** lowSpace
Scope: Enterprise
Subobject: Irrelevant
Node: Irrelevant
Property: hpws, which is contained in hp124's property group

The trigger and transition fail the property rule.

- **Name:** lowSpace
Scope: Subobject
Subobject: ifEntry.2
Node: hp125
Property: includeMe

The trigger and transition fail the node rule.

- **Name:** lowSpace
Scope: Subobject
Subobject: Unassigned (transition from Ground)
Node: Unassigned (transition from Ground)
Property: NO_PROP

The trigger and transition fail the subobject rule.

Example 3

A trigger named lowSpace with the subobject \$NULL, the node \$ANY, and the property NO_PROP *would* prompt the following transitions:

- **Name:** lowspace
Scope: Node
Subobject: Irrelevant
Node: hp125
Property: includeMe
- **Name:** lowspace
Scope: Enterprise
Subobject: Irrelevant
Node: Irrelevant
Property: dontIncludeMe

The lowSpace trigger *would not* prompt the following transitions:

- **Name:** lowspace
Scope: Subobject
Subobject: ifEntry.2
Node: hp125
Property: includeMe

The trigger and transition fail the subobject rule.

- **Name:** lowspace
Scope: Subobject
Subobject: Any string at all, including the empty string
Node: Any node at all
Property: NO_PROP

The trigger and transition fail the subobject rule.

Auditing Behavior Models

NerveCenter includes an auditing feature that looks for:

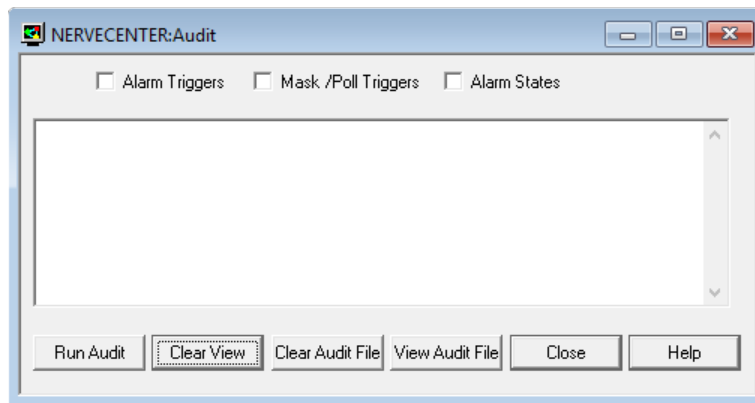
- Alarm transitions for which there are no corresponding triggers
- Triggers that are fired by a poll or a mask and are not used in alarms
- Alarms with states that are unreachable

You should audit your database periodically to ensure that you don't have extraneous objects in your database and that alarms you're currently using don't have unreachable states or unusable transitions.

TO PERFORM AN AUDIT

1. Choose **Audit** from the client's **Admin** menu.

The Audit window appears.



2. Check one or more of the checkboxes above the text area.
 - Checking the **Alarm Triggers** checkbox indicates that you want to see information about alarm transitions for which there are no corresponding triggers.
 - Checking the **Mask /Poll Triggers** checkbox indicates that you want to see information about polls and masks that fire triggers that are not used by any currently defined alarm.
 - Checking the **Alarm States** checkbox indicates that you want to see information about alarms that contain states that are unreachable.
3. Select the **Run Audit** button.

The results of the audit are written to the text area in the Audit window and to **/opt/OSInc/userfiles/logs**.

The other buttons in the Audit window have the following functions:

- **Clear** clears the contents of the text area in the Audit window.
- **Clear Audit File** clears the contents of the file audit.txt.
- **View Audit File** displays the contents of the file audit.txt in the text area of the Audit window.

Behavior Model Log

NerveCenter logs most behavior model changes to `/opt/OSInc/userfiles/logs/AuditTrail.log`. [Table 28](#) describes the events recorded in `AuditTrail.log`. You can use this to help trace how and when a behavior model stops functioning.

Table 28: Events Recorded in AuditTrail.log

NerveCenter Object	Events Recorded
Alarm	<ul style="list-style-type: none"> ■ Add alarm ■ Update alarm ■ Delete alarm ■ Alarm On/Off
Poll	<ul style="list-style-type: none"> ■ Add poll ■ Update poll ■ Delete poll ■ Poll On/Off
Mask	<ul style="list-style-type: none"> ■ Add mask ■ Update mask ■ Delete mask ■ Mask On/Off
Perl Subroutine	<ul style="list-style-type: none"> ■ Add Perl Subroutine ■ Update Perl Subroutine ■ Delete Perl Subroutine
Property Group	<ul style="list-style-type: none"> ■ Add/Delete Property Group

The following is a sample AuditTrail.log.

```

01/21/2004 17:08:54 Wed - User bxie added Property: newprop1
01/21/2004 17:08:54 Wed - User bxie added Property: newprop
01/21/2004 17:08:54 Wed - User bxie added PropertyGroup: newGroup
01/21/2004 17:09:17 Wed - User bxie added Property: newprop2
01/21/2004 17:09:17 Wed - User bxie added PropertyGroup: newGroup1
01/21/2004 17:09:29 Wed - User bxie updated PropertyGroup:
newGroup
01/21/2004 17:11:04 Wed - User root updated PropertyGroup:
newGroup
01/21/2004 17:12:14 Wed - User bxie added poll with name of
newPoll
01/21/2004 17:12:24 Wed - User bxie updated poll with name of
newPoll
01/21/2004 17:12:35 Wed - User bxie turned Off poll with name of
newPoll
01/21/2004 17:12:46 Wed - User bxie deleted poll with name of
newPoll
01/21/2004 17:13:43 Wed - User root added mask with name of
newMask

```


01/21/2004 17:13:57 Wed - User root updated mask with name of newMask
01/21/2004 17:14:10 Wed - User bxie turned Off mask with name of newMask
01/21/2004 17:14:25 Wed - User bxie deleted mask with name of newMask
01/21/2004 17:15:07 Wed - User root added alarm with name of NewAlarm
01/21/2004 17:15:41 Wed - User root updated alarm with name of NewAlarm
01/21/2004 17:55:57 Wed - User bxie has turned Off the alarm with name of NewAlarm
01/21/2004 17:56:03 Wed - User bxie has turned On the alarm with name of NewAlarm
01/21/2004 17:56:03 Wed - User bxie has turned On the alarm with name of newAlarm
01/21/2004 17:56:03 Wed - User bxie has turned On the alarm with name of newAlarm2
01/21/2004 17:56:03 Wed - User bxie has turned On the alarm with name of newAlarm3
01/21/2004 17:56:08 Wed - User bxie deleted alarm with name of NewAlarm
01/21/2004 17:56:08 Wed - User bxie deleted alarm with name of newAlarm
01/21/2004 17:56:09 Wed - User bxie deleted alarm with name of newAlarm2
01/21/2004 17:56:10 Wed - User bxie deleted alarm with name of newAlarm3
01/21/2004 17:56:33 Wed - User bxie added perl subroutine with name of nEwPerl
01/21/2004 17:56:58 Wed - User bxie updated perl subroutine with name of newPerl
01/21/2004 17:57:12 Wed - User bxie updated perl subroutine with name of newPerl
01/21/2004 17:57:24 Wed - User bxie deleted perl subroutine with name of newPerl
01/21/2004 17:57:46 Wed - User bxie added opcmask with name of newOpcMask
01/21/2004 17:57:51 Wed - User bxie updated opcmask with name of newOpcMask
01/21/2004 17:57:58 Wed - User bxie turned Off opcmask with name of newOpcMask
01/21/2004 17:58:20 Wed - User bxie deleted opcmask with name of newOpcMask
01/21/2004 17:58:39 Wed - User bxie removed Property: newprop2
01/21/2004 17:58:39 Wed - User bxie removed Property: newprop1

```
01/21/2004 17:58:39 Wed - User bxie removed Property: newprop
01/21/2004 17:58:39 Wed - User bxie removed PropertyGroup:
newGroup1
01/21/2004 17:58:39 Wed - User bxie removed PropertyGroup:
newGroup
```

The NerveCenter downstream alarm suppression behavior model monitors nodes in a complex network. Using topology information, the model uses the relationships between nodes to determine the status of those nodes accurately. You can also use the model to log data to the database for outage and availability reports.

This appendix describes how the model works, how to test the model, and the technical details of the model. The latest downstream alarm suppression model, `nodestatus_dwnstrm.mod`, is included with the current release of LogMatrix NerveCenter. You can also get them from the LogMatrix NerveCenter site at <http://www.logmatrix.com>.

This appendix includes the following sections:

- [Understanding How the Model Works below](#)
- [Testing the Model on page 387](#)
- [Understanding the Technical Details on page 390](#)

Understanding How the Model Works

The first downstream alarm suppression model (which included `DSCollectRoutes`, `DSIcmpStatus`, and `DSSnmpStatus`), used information about local routers to determine the status of an unreachable node. If a route existed for the node, the node was assumed to be down; otherwise, it was marked as unreachable. In either case, the node was suppressed. For simple networks that consisted of nodes behind routers, this model was adequate. However, for more complex networks with multiple routers, switches, and hubs, and for certain routing protocols, the new model provides a more accurate determination of a node's status.

What is a complex network, as opposed to a simple network? A simple network might include single parent-child relationships. Nodes that are dependent on other nodes for a route to the NerveCenter server are *child* nodes. Nodes on which other nodes are dependent are *parent* nodes.

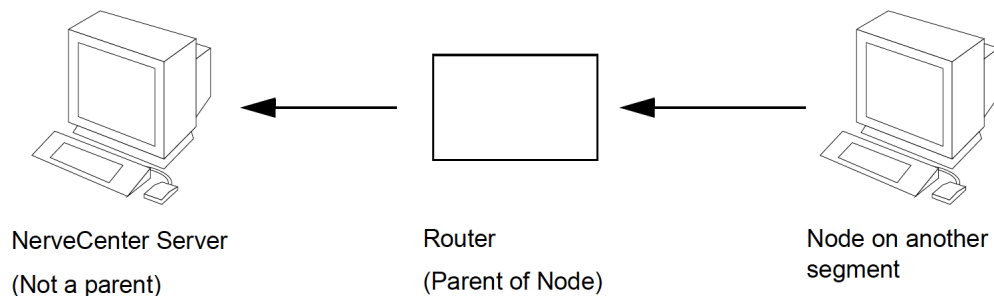


Figure 64: A Simple Network

Note: The NerveCenter Server is not a parent node, nor does it have a parent. Logically, then, no node on the same segment as the NerveCenter Server has a parent.

A more complex network might include nodes with multiple parents and nodes that are themselves parents to other nodes.

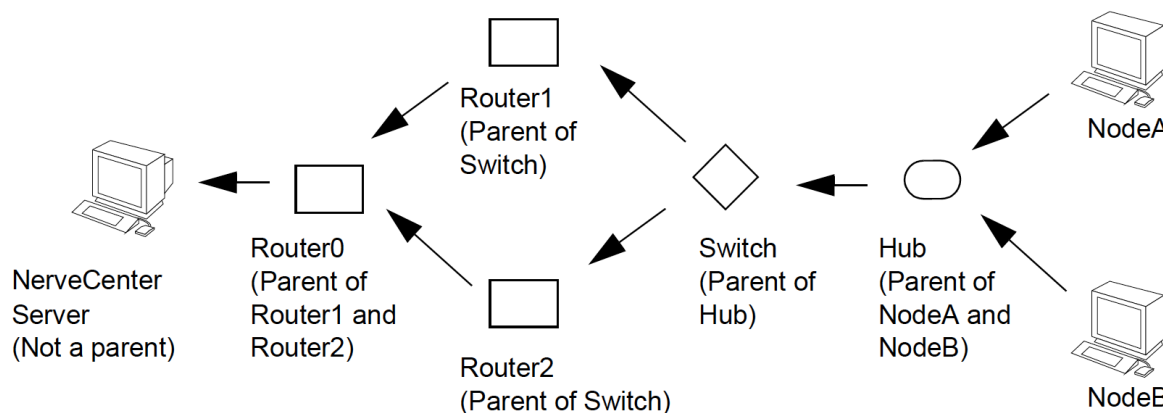


Figure 65: A More Complex Network

The new model uses the status of devices between NerveCenter and managed nodes in the network to make real-time determinations about whether nodes are up, down, or unreachable. NerveCenter can then take appropriate actions based on the statuses of those nodes. For example, suppose NerveCenter is monitoring 1000 nodes, and 300 nodes behind a router stop responding to polls. NerveCenter can use the status of the router and any intermediate devices to determine whether the nodes are down or unreachable. If the nodes are actually down, NerveCenter forwards the appropriate alarms to the network management platform; however, if they are unreachable, NerveCenter just forwards one critical alarm for the router and uses built-in LogMatrix NerveCenter Smart Polling technology to stop suppressible polls for those nodes until they are available again.

NerveCenter can get information about the nodes in the following ways:

- You create a text file that defines the relationships of the nodes on your network. Then, you create an alarm that uses a Perl subroutine with some built-in functions to load that information into the database.

Note: Don't include nodes that have many routes (nodes that have hundreds of parents, for example). The overhead necessary to maintain parent information about these nodes is unnecessary because the likelihood that every single route to the nodes is going to be down is very small.

Once NerveCenter has that relationship information, the `DwnStrmSnmpStatus` and `DwnStrmIcmpStatus` alarms monitor nodes and maintain their statuses in the NerveCenter database.

Note: The accuracy of NerveCenter decisions depends on the accuracy of the topology information it receives. Note that if you export node information that includes parent information, from one server to another on a different segment, the parent data might not be accurate because the topology perspective will be different.

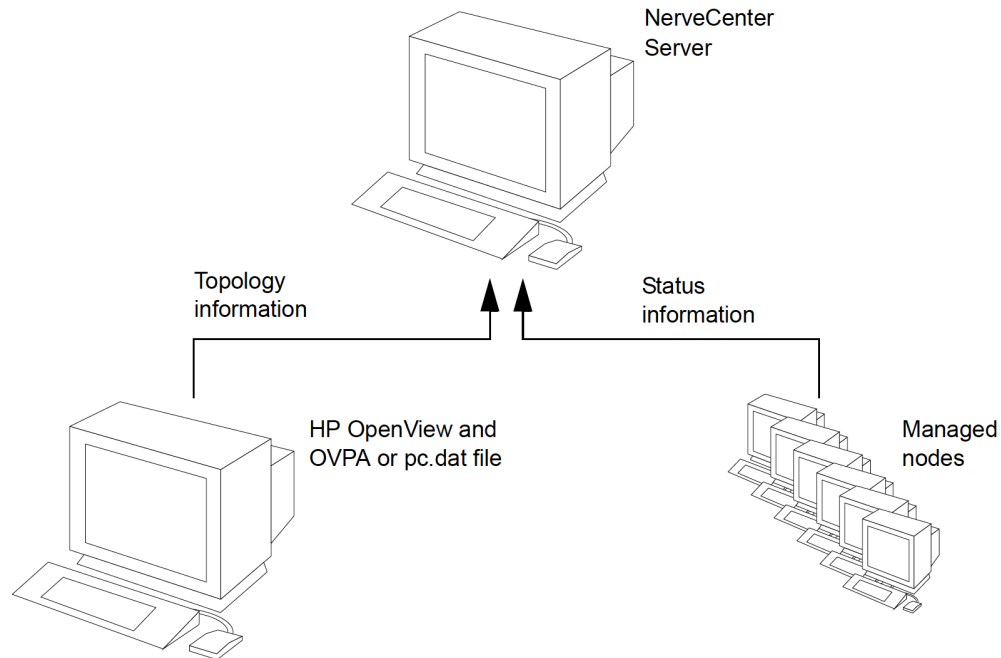


Figure 66: NerveCenter Maintains Parent-Child Relationship and Status Information

Nodes can have the following statuses: up, testing, down, and unreachable. Any node that responds has a status of up. The first time a node does not respond, its status is set to testing. While a node is in testing, its status is not updated again until NerveCenter determines that the node is up, down, or unreachable.

NerveCenter decides whether the node is down or unreachable based on whether the node has parents, whether the parents' statuses are more current than the node's last status update, and what those statuses are.

Note: The current downstream alarm suppression model evaluates parent status at the node level, not the subobject or interface level.

The model uses the following logic:

- The node is set to unreachable if the following condition is true: all parents have more current statuses and no parents are up or in testing.
- The node is set to down if one of the following conditions is true:
 - At least one parent has a more current status than the node and is up
 - The node has no parents
- The status of the node does not change as long as one of the following is true:
 - No parents have a more current status than the node
 - One or more parents have a more current status than the node but are not up

The key is to only update a node's status when NerveCenter can make a definitive decision about the status based on real-time data, which can only happen when the parents' status is more current than the node's status.

If the node does go to down or unreachable, NerveCenter continues to monitor the node and its parents to determine if the node is available again, if the parents' statuses have affected the status of the node, or if there has been no change.

For example, the [Figure 67](#) shows a node that has one parent. At T_0 , the node does not respond to an SNMP poll, so the alarm transitions from ground to error and the node's status is updated to testing. If the node does not respond to a second poll at T_1 , the alarm transitions from error to testing but the node's status is not updated. On a circular transition that loops back to the testing state, a Perl subroutine checks—and continues to check—the parent's status. At T_2 , the parent's status has been updated. Since the parent's status is more current than the node's status, the alarm transitions to unreachable and the node's status is set to unreachable. At T_3 , the parent's status has not changed, so the node's status is not updated.

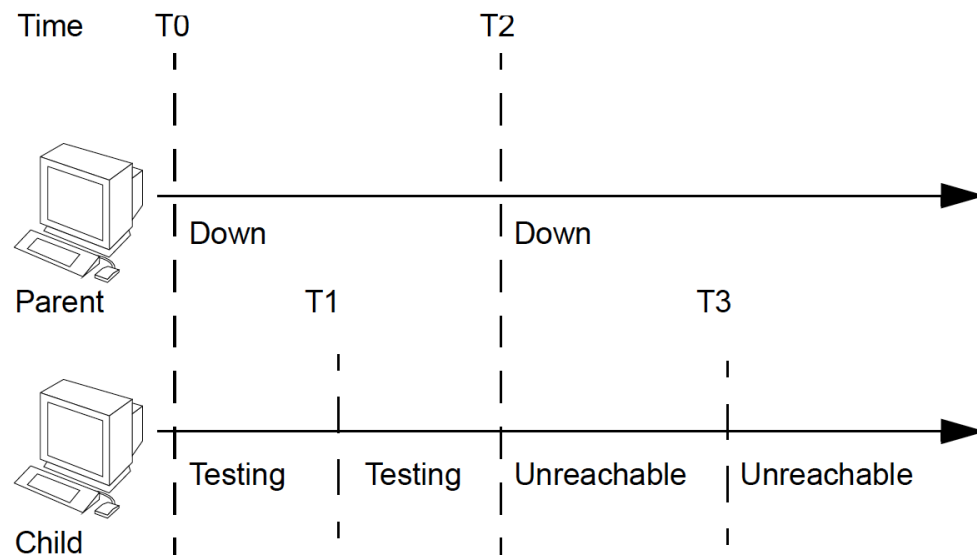


Figure 67: Updating a Node's Status Depends on When the Parent's Status Was Last Updated

As long as the parent's status remains down and is more current than the node's status every time the Perl subroutine checks it, the node's status is refreshed.

For more details about the new downstream alarm suppression model, see [Understanding the Technical Details on page 390](#).

Testing the Model

The alarm suppression model is based on this concept: by monitoring whether nodes are dependent on other nodes (parent-child relationships) and by keeping each node's status updated proactively, the model can make accurate assessments as to what the statuses of dependent, or child, nodes are.

The following sections describe how to test the models:

- [Importing the New Model below](#)
- [Identifying Parent-Child Relationships on the next page](#)
- [Making the Relationship Information Available to NerveCenter on the next page](#)
- [Testing the Alarm Suppression Model on page 389](#)

Importing the New Model

The alarm suppression model is not included in the default database that is installed with NerveCenter. You must import the model before you can use it. The model includes all of the objects you need, including the alarms, polls, masks, Perl subroutines, and so on.

Caution: Older versions of the imported objects will be overwritten (for example, IcmpPoll).

TO IMPORT THE NEW MODEL

1. Start the Client and connect to the NerveCenter Server.
2. From the **Server** menu, select **Import Objects and Nodes**.
3. Select **Browse**.
4. Double-click the **node_status** directory.
5. Select **nodestatus_dwnstrm.mod** and then **Open**.
6. Select **OK**.

A message is displayed when the file has been imported. The following alarms are listed in the alarm list:

- DwnStrmIcmpStatus
- DwnStrmIcmpStatus_LogToDB
- DwnStrmSnmpStatus
- DwnStrmSnmpStatus_LogToDB

See [Importing and Exporting NerveCenter Nodes and Objects on page 353](#) for complete details on importing models.

Identifying Parent-Child Relationships

In order to use NerveCenter's Downstream Alarm Suppression behavior model, it is necessary to establish the parent-child relationship between nodes. To identify parent-child relationships manually

1. Open a new text file.
2. Include a line for each node that has parents. Use the following syntax:

```
child parent
```

where *child* is the name of the node and *parent* is the name of each node on which the child is dependent. If you have more than one parent, separate parents by typing a space between each one.

Note: If NerveCenter uses a full domain name for the node, use the full name in this file when referring to that node.

For example, if nodeA is dependent on nodeB.domain.com and nodeC, and nodeB.domain.com is dependent on nodeD, then the text file would look like this:

```
nodeA nodeB.domain.com nodeC
nodeB.domain.com nodeD
```

3. Save and close the file.

The name and location of the file do not matter, as long as you remember the location to use in the next procedure.

Making the Relationship Information Available to NerveCenter

If you created a text file with the relationship information, you must load that information into NerveCenter.

TO LOAD RELATIONSHIP INFORMATION INTO NERVECENTER

1. In the NerveCenter Client, create an alarm that you can transition on demand. On the transition, call a Perl subroutine that includes the following function:

```
NC::LoadParentsFromFile ( FileName );
```

where *FileName* is the name of the file you created.

2. Transition the alarm. After the alarm transitions, you can turn the alarm off.

Caution: If you modify the file, you must repeat this procedure.

To make sure the contents of the file were read correctly, you can create another alarm with a Perl subroutine that includes the following function:

```
NC::DumpParentsToFile (FileName);
```

The information will be written to the file on the local machine.

To remove relationship information, you can create an alarm with a Perl subroutine that includes the following function:

```
NC::RemoveAllParents ();
```

Testing the Alarm Suppression Model

You can test the model by turning the `DwnStrmSnmpStatus` alarm (see [DwnStrmSnmpStatus Alarm on page 392](#)) and `DwnStrmlcmpStatus` alarm (see [DwnStrmlcmpStatus Alarm on page 398](#)) on, and then simulating a node being unreachable.

TO TEST THE ALARM SUPPRESSION MODEL

1. Make sure the Client is connected to the NerveCenter Server.
2. From the **Admin** menu, select **Alarm Definition List**.
The Alarm Definition List dialog is displayed.
3. In the listbox, right-click on `DwnStrmSnmpStatus` and select **On**.
4. In the listbox, right-click on `DwnStrmlcmpStatus` and select **On**.

The `DwnStrmSnmpStatus` and `DwnStrmlcmpStatus` alarms monitor the status of managed nodes.

Note: To use the model to log data against which you can run availability reports, use `DwnStrmSnmpStatus_LogToDB` and `DwnStrmlcmpStatus_LogToDB` instead of the versions that don't log data (`DwnStrmSnmpStatus` and `DwnStrmlcmpStatus`).

To simulate a node being unreachable, you can change the IP address of an existing node or that of a new node to an invalid address for your network. (For example, you might use 10.10.10.10.) If you have a test network available, you can also make nodes unreachable by unplugging devices—a router, for example.

NerveCenter detects errors since the node no longer responds to polls. As a result, NerveCenter reevaluates and updates the node status. If the alarm is in an `AgentDown`, `DeviceDown`, or `Unreachable` state, NerveCenter suppresses suppressible alarms for that node until it is available again.

To make sure the statuses of the nodes are correct, you can create an alarm with a Perl subroutine that includes the following function:

```
NC::DumpNodeStatusToFile (FileName);
```

The information will be written to the file in the NerveCenter installation directory on the local machine.

Understanding the Technical Details

The two particular areas of interest in the model are the alarms used to monitor device status and the Perl subroutines used to store and evaluate relationship and status information. See the following sections for details about those types of objects:

- [Alarms on the facing page](#)
- [Perl Subroutines on page 402](#)

The following objects are imported when you import the MOD file.

Caution: Older versions of the imported objects will be overwritten (for example, SnmpPoll).

- Alarms
 - DwnStrmIcmpStatus (off)
 - DwnStrmIcmpStatus_LogToDB (off)
 - DwnStrmSnmpStatus (off)
 - DwnStrmSnmpStatus_LogToDB (off)
- Properties
 - icmpStatus
 - nl-ping
 - system
- Polls
 - IS_IcmpPoll (on)
 - IS_IcmpFastPoll (on)
 - SnmpFastPoll (on)
 - SnmpPoll (on)
 - SS_IcmpPoll (on)
 - SS_IcmpFastPoll (on)
- Masks
 - ColdStart (on)
 - WarmStart (on)
- Triggers
 - agentUp
 - agentUpFast
 - warmStart
 - coldStart
 - Down
 - ICMP_ERROR

- IS_Icmp_Error
- ISF_Icmp_Error
- ISnodeUpFast
- ISnodeUp
- SNMP_TIMEOUT
- SS_Icmp_Error
- SS_PortUnreachTesting
- SSF_Icmp_Error
- SSnodeUpFast
- SSnodeUp
- UnReachable
- Severities
 - Critical
 - Inform
 - Minor
 - Normal
- Perl subroutines
 - SetNodeStatusDown
 - SetNodeStatusTesting
 - SetNodeStatusUnReachable
 - SetNodeStatusUp
 - SS_IcmpError
 - TestParentSetNode
 - TestParentStatus

Alarms

The downstream alarm suppression behavior model monitors node status using both SNMP and ICMP. This section includes descriptions of the following alarms:

- [DwnStrmSnmpStatus Alarm on the next page](#)
- [DwnStrmIcmpStatus Alarm on page 398](#)

Note: You must import the downstream behavior models before they become available in NerveCenter Client. From the Server menu in Client, choose **Import Objects and Nodes**, and then browse to select the node status models.

DwnStrmSnmpStatus Alarm

This alarm accurately monitors the status of nodes and their SNMP agents by taking into consideration the status of the nodes' parents. This alarm is the same as the DwnStrmSnmpStatus_LogToDB version, except that the DwnStrmSnmpStatus_LogToDB version also logs data on most transitions.

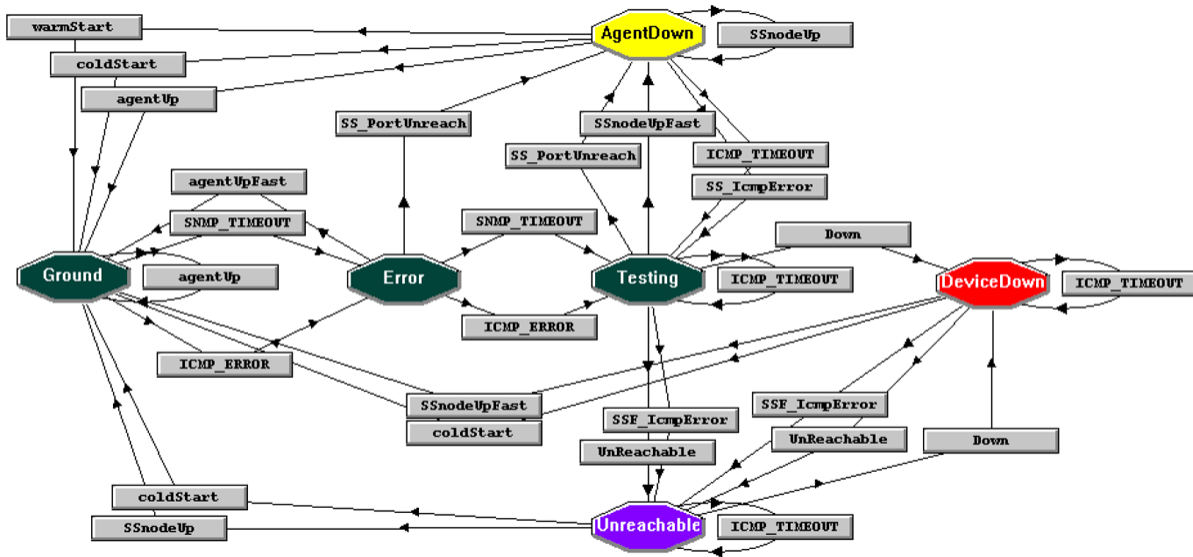


Figure 68: DwnStrmSnmpStatus/DwnStrmSnmpStatus_LogToDB Alarm State Diagram

Table 29 lists the severity of each state:

Table 29: Severities of Each State in DwnStrmSnmpStatus

State	Severity	Color
Ground	Normal	Green
Error	Normal	Green
Testing	Normal	Green
AgentDown	Minor	Yellow
DeviceDown	Critical	Red
Unreachable	Inform	Purple

When this alarm is turned on, the following polls and masks cause state transitions:

- ColdStart (trap mask)
- SnmpFastPoll (SNMP get request)
- SnmpPoll (SNMP get request)
- SS_IcmpFastPoll (ICMP echo request, or ping)
- SS_IcmpPoll (ICMP echo request, or ping)
- WarmStart (trap mask)

This alarm uses the following Perl subroutines:

- [SS_IcmpError Perl Subroutine on page 402](#)
- [SetNodeStatus Perl Subroutines on page 403](#)
- [TestParentStatus Perl Subroutine on page 404](#)
- [TestParentSetNode Perl Subroutine on page 407](#)

Note: Before turning this alarm on, NerveCenter must have loaded the relationship data. See [Identifying Parent-Child Relationships on page 388](#) and [Making the Relationship Information Available to NerveCenter on page 388](#).

The following sections describe the states in the DwnStrmSnmpStatus alarm and the transitions and actions that can happen from those states:

- [Ground State below](#)
- [Error State on the next page](#)
- [Testing State on the next page](#)
- [AgentDown State on page 396](#)
- [Unreachable State on page 396](#)
- [DeviceDown State on page 397](#)

Ground State

In Ground state, the node is reachable and the SNMP agent is up.

As long as the node and agent respond to the SnmpPoll and SnmpFastPoll requests, the agentUp circular transition is triggered. The agentUp transition calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to refresh the update time.

If the node does not respond to the polls, the following triggers can transition the alarm from Ground to Error:

- ICMP_ERROR
- SNMP_TIMEOUT

Transitions to the Error state call the SetNodeStatusTesting Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status to Testing.

ICMP_ERROR also calls the SS_IcmpError Perl subroutine (see [SS_IcmpError Perl Subroutine on page 402](#)). If the SS_IcmpError Perl subroutine determines that the port is unreachable, it fires SS_PortUnreach. The SS_PortUnreach trigger does the following:

- Transitions the alarm to an AgentDown state
- Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the agent is down
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

Error State

The alarm suppression behavior model uses the Error state to confirm that there is actually a problem (as opposed to a dropped packet, for example). From the Error state, a node can transition back to Ground, to Testing, or to AgentDown.

If the node and agent respond to the SnmpFastPoll request, the agentUpFast transition is triggered. The agentUpFast transition does the following:

- Returns the alarm to Ground state
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If the node still does not respond to the poll, the following triggers transition the alarm from Error to Testing:

- ICMP_ERROR, which calls the SS_IcmpError Perl subroutine (see [SS_IcmpError Perl Subroutine on page 402](#))
- SNMP_TIMEOUT

If the SS_IcmpError Perl subroutine determines that the port is unreachable, it fires SS_PortUnreach. The SS_PortUnreach trigger does the following:

- Transitions the alarm to an AgentDown state
- Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the agent is down
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

Testing State

While an alarm is in the Testing state, NerveCenter identifies whether the node is:

- Down
- Unreachable
- Up, but its agent is down

If SS_PortUnreach is triggered by the SS_IcmpError Perl subroutine while the node transitioned from Error to Testing or if SS_nodeUpFast is triggered by SS_IcmpFastPoll, the trigger:

- Transitions the alarm to an AgentDown state
- Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the agent is down
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If `SSF_IcmpError` is triggered by `SSF_IcmpFastPoll`, the trigger:

- Transitions the alarm to an Unreachable state
- Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the node is unreachable
- Calls the `SetNodeStatusUnreachable` Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If `SS_nodeUpFast` results in a circular `ICMP_TIMEOUT` transition, the `TestParentStatus` Perl subroutine (see [TestParentStatus Perl Subroutine on page 404](#)) looks up the status of the parents. If `TestParentStatus` can determine the node's state based on the parents' status, `TestParentStatus` fires the appropriate trigger: `UnReachable` or `Down`.

- The `UnReachable` trigger:
 - Transitions the alarm to an Unreachable state
 - Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the node is unreachable
 - Calls the `SetNodeStatusUnreachable` Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately
- The `Down` trigger:
 - Transitions the alarm to a `DeviceDown` state
 - Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the node is unreachable
 - Calls the `SetNodeStatusDown` Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately
 - Sends an Inform action to notify a network management platform or another NerveCenter of the status of this node

AgentDown State

While an alarm is in the AgentDown state, NerveCenter continues to monitor the node for changes. As long as the node responds to the SS_IcmpPoll requests, the SSnodeUp transition is triggered. The SSnodeUp transition calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to refresh the update time.

If the node does not respond to the polls, the following triggers transition the node from AgentDown to Testing:

- ICMP_TIMEOUT
- SS_IcmpError

Each transition calls the SetNodeStatusTesting Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status to Testing.

If NerveCenter receives a warmStart trap or a coldStart trap, or agentUp is triggered in response to an SnmpPoll response, the trigger:

- Transitions the alarm to a Ground state
- Uses the Set Attribute action to turn poll suppression off so NerveCenter can resume all normal polling
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the time of the last status change so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

Unreachable State

While an alarm is in the Unreachable state, NerveCenter continues to monitor the node for any changes. If NerveCenter receives a coldStart trap or SSnodeUp is triggered by a response to SS_IcmpPoll, the trigger:

- Transitions the alarm to a Ground state
- Uses the Set Attribute action to turn poll suppression off so NerveCenter can resume all normal polling
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If the poll does not get a response and an ICMP_TIMEOUT transition is triggered, NerveCenter calls the TestParentSetNode Perl subroutine (see [TestParentSetNode Perl Subroutine on page 407](#)), which looks up the parent status. If TestParentSetNode can determine the node's state based on the parents' status, TestParentSetNode fires the Down trigger or refreshes the node's update time.

The Down trigger:

- Transitions the alarm to a DeviceDown state
- Calls the SetNodeStatusDown Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately
- Sends an Inform action to notify a network management platform or another NerveCenter of the status of this node

DeviceDown State

While an alarm is in the DeviceDown state, NerveCenter continues to monitor the node for any changes.

If NerveCenter receives a coldStart trap or the SSnodeUpFast transition is triggered by an SS_IcmpFastPoll, the trigger:

- Transitions the alarm to a Ground state
- Uses the Set Attribute action to turn poll suppression off so NerveCenter can resume all normal polling
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the time of the last status change so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If SSF_IcmpError is triggered by SS_IcmpFastPoll, the trigger:

- Transitions the alarm to an Unreachable state
- Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the node is unreachable
- Calls the SetNodeStatusUnreachable Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If the poll does not get a response and a circular ICMP_TIMEOUT transition is triggered, NerveCenter calls the TestParentSetNode Perl subroutine (see [TestParentSetNode Perl Subroutine on page 407](#)), which looks up the status of the parents. If TestParentSetNode can determine the node's state based on the parents' status, TestParentStatus fires the Unreachable trigger or refreshes the node's update time.

The Unreachable trigger:

- Transitions the alarm to an Unreachable state
- Calls the SetNodeStatusUnReachable Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

The following sections describe the states in the `DwnStrmlcmpStatus` alarm and the transitions and actions that can happen from those states:

- [Ground State below](#)
- [Error State below](#)
- [Testing State on the next page](#)
- [Unreachable State on page 401](#)
- [DeviceDown State on page 401](#)

Ground State

In Ground state, the node is reachable.

As long as the node responds to the `IS_IcmpPoll` requests, the `ISnodeUp` transition is triggered. The `ISnodeUp` transition calls the `SetNodeStatusUp` Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to refresh the update time.

If the node does not respond to the polls, the following triggers can transition the alarm from Ground to Error:

- `ICMP_TIMEOUT`
- `IS_IcmpError`

Transitions to the Error state call the `SetNodeStatusTesting` Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status to Testing.

Error State

The alarm suppression behavior model uses the Error state to confirm that there is actually a problem (as opposed to a dropped packet, for example). From the Error state, an alarm can transition back to Ground or to Testing.

If the node responds to the `IS_IcmpFastPoll` request, the `ISnodeUpFast` transition is triggered. The trigger:

- Returns the alarm to Ground state
- Calls the `SetNodeStatusUp` Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If the node still does not respond to the poll, the following triggers transition the alarm from Error to Testing:

- `ICMP_TIMEOUT`
- `ISF_IcmpError`

Testing State

While an alarm is in the Testing state, NerveCenter identifies whether the node is down or unreachable. If ISnodeUpFast is triggered in response to an IS_IcmpFastPoll poll while the node is in the Testing state, the trigger:

- Transitions the alarm to Ground
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If ISF_IcmpError is triggered in response to an IS_IcmpFastPoll poll while the node is in the Testing state, the trigger:

- Transitions the alarm to Unreachable
- Calls the SetNodeStatusUnreachable Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If ISnodeUpFast results in a circular ICMP_TIMEOUT transition, NerveCenter calls the TestParentStatus Perl subroutine (see [TestParentStatus Perl Subroutine on page 404](#)) looks up the status of the parents. If TestParentStatus can determine the node's state based on the parents' status, TestParentStatus fires the appropriate trigger: UnReachable or Down.

- The UnReachable trigger:
 - Transitions the alarm to an Unreachable state
 - Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the node is unreachable
 - Calls the SetNodeStatusUnreachable Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately
- The Down trigger:
 - Transitions the alarm to a DeviceDown state
 - Uses the Set Attribute action to suppress the node so the node won't be polled by suppressible polls while the node is unreachable
 - Calls the SetNodeStatusDown Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately
 - Sends an Inform action to notify a network management platform or another NerveCenter of the status of this node

Unreachable State

While an alarm is in the Unreachable state, NerveCenter continues to monitor the node for any changes. If ISnodeUp is triggered by a response to IS_IcmpPoll, the trigger:

- Transitions the alarm to a Ground state
- Uses the Set Attribute action to turn poll suppression off so NerveCenter can resume all normal polling
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If the poll does not get a response and a circular ICMP_TIMEOUT transition is triggered, NerveCenter calls the TestParentSetNode Perl subroutine (see [TestParentSetNode Perl Subroutine on page 407](#)), which looks up the status of the parents. If TestParentSetNode can determine the node's state based on the parents' status, TestParentSetNode either fires the Down trigger or refreshes the node's update time.

The Down trigger:

- Transitions the alarm to a DeviceDown state
- Calls the SetNodeStatusDown Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately
- Sends an Inform action to notify a network management platform or another NerveCenter of the status of this node

DeviceDown State

While an alarm is in the DeviceDown state, NerveCenter continues to monitor the node for any changes. If the ISnodeUpFast transition is triggered by an IS_IcmpFastPoll, the trigger:

- Transitions the alarm to a Ground state
- Uses the Set Attribute action to turn poll suppression off so NerveCenter can resume all normal polling
- Calls the SetNodeStatusUp Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the time of the last status change so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If ISF_IcmpError is triggered in response to an IS_IcmpFastPoll poll while the node is in the Testing state, the trigger:

- Transitions the alarm to Unreachable
- Calls the SetNodeStatusUnreachable Perl subroutine (see [SetNodeStatus Perl Subroutines on page 403](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

If the poll does not get a response and a circular ICMP_TIMEOUT transition is triggered, the TestParentSetNode Perl subroutine (see [TestParentSetNode Perl Subroutine on page 407](#)), which looks up the status of the parents. If TestParentSetNode can determine the node's state based on the parents' status, TestParentSetNode fires the Unreachable trigger or refreshes the node's update time.

The Unreachable trigger:

- Transitions the alarm to an Unreachable state
- Calls the SetNodeStatusUnReachable Perl subroutine (see [TestParentStatus Perl Subroutine on page 404](#)) to update the status so that NerveCenter can evaluate the states of the children of this node, if there are any, accurately

Perl Subroutines

The new downstream alarm suppression behavior model uses several Perl subroutines to store parent-child relationships and maintain node statuses. This section includes descriptions of the following Perl subroutines:

- [SS_IcmpError Perl Subroutine below](#)
- [SetNodeStatus Perl Subroutines on the facing page](#)
- [TestParentStatus Perl Subroutine on page 404](#)
- [TestParentSetNode Perl Subroutine on page 407](#)

SS_IcmpError Perl Subroutine

The ICMP_ERROR transition calls this Perl subroutine to evaluate the error and determine whether or not it indicates that the node is unreachable. If the ICMP error is Port Unreachable, the node is up and reachable. It is assumed that other ICMP errors indicate an unreachable node. This assumption may be incorrect depending on the behavior of your network. To include other ICMP errors that indicate that the node is unreachable, modify this Perl subroutine.

```
my $Type = VbValue( 0 );
my $Code = VbValue( 1 );
if( $Type == 3 && $Code == 3 )
{
    FireTrigger( "SS_PortUnreach" );
}
else
{
    # Modify this else to eliminate other types of
    # ICMP errors that are not indicative of an
    # unreachable node. The assumption is that if
    # SS_IcmpError is fired, we are being told, by
    # the network, that the node is unreachable.
    FireTrigger( "SS_IcmpError" );
}
```

SetNodeStatus Perl Subroutines

For the DwnStrmSnmpStatus and DwnStrmIcmpStatus alarms, all state transitions—except transitions from Error to Testing—call one of the following Perl subroutines:

- SetNodeStatusTesting
- SetNodeStatusDown
- SetNodeStatusUnreachable
- SetNodeStatusUp

These Perl subroutines update the node status so the node's children can accurately update their statuses based on the node's status.

SetNodeStatusTesting

```
my $Return;
$return = NC::SetNodeStatus($NodeName, "Testing");
#If $Return = 0, operation failed
```

SetNodeStatusDown

```
my $Return;
$return = NC::SetNodeStatus($NodeName, "Down");
#If $Return = 0, operation failed
```

SetNodeStatusUnreachable

```
my $Return;
$return = NC::SetNodeStatus($NodeName, "Unreachable");
#If $Return = 0, operation failed
```

SetNodeStatusUp

```
my $Return;
$return = NC::SetNodeStatus($NodeName, "Up");
#If $Return = 0, operation failed
```

TestParentStatus Perl Subroutine

For the `DwnStrmSnmpStatus` and `DwnStrmIcmpStatus` alarms, if a node is in a Testing state, the ERROR trigger is fired every time the node is polled and doesn't respond. Each resulting ERROR transition calls the `TestParentStatus` Perl subroutine.

The `TestParentStatus` Perl subroutine tests the parent node status and determines the status of the node by doing the following:

- If the node has parents, `TestParentStatus` evaluates each parent's last update time. Based on the following rules, `TestParentStatus` sets a flag (`TriggerFlag`) that determines what trigger, if any, should be fired.
 - If no parents have an update time more recent than the node's update time, then `TriggerFlag` is set to Testing.
 - If at least one parent has a more recent update time but is not up, the flag is set to Testing.
 - If at least one parent has a more recent update time and is up, the flag is set to Down, regardless of the status or time of last update of any other parent.
 - If all parents have more recent update times and no parent is up or in testing, the flag is set to Unreachable.
- If the node has no parents, `TriggerFlag` is set to Down.

If `TriggerFlag` is set to Testing, `TestParentStatus` does nothing because `TestParentStatus` must have more information to make an accurate decision. If the alarm should be in another state, `TestParentStatus` fires the appropriate trigger to transition the node into that state.

The code for this subroutine follows:

```
# The purpose of this subroutine is to test the parent
# node status and fire the appropriate trigger to take the
# alarm to either down or unreachable. You must make sure
# that all parents are being monitored with the status
# alarms.
use NC;
my $NodeUpdateTime; # Last time node status was updated
my $LastNodeStatus; # Last node status
my @Parents = (); # Array of parents
my $Parent; # Parent Node
my $ParentUpdateTime; # Last time parent node status was updated
my $ParentStatus; # Last parent status
my $TriggerFlag = "NotSet";
my $ParentNotUpdated = 0; # Remember if we have any parents not
updated

#Define all triggers that can be fired
DefineTrigger('UnReachable');
DefineTrigger('Down');
DefineTrigger('Testing');

# Get the last node status and update time for this node
($LastNodeStatus,$NodeUpdateTime) = NC::GetNodeStatus($NodeName);
```



```
# Get the array of parents for this node
@Parents = NC::GetParents($nodeName);
if( defined( $Parents[0] ) )
{
  # Test each parent, if ANY are ok, we assume the node
  # is reachable. Parents update time must be past the
  # last time the node was updated or we can't assume the
  # status is accurate.
  foreach $Parent (@Parents)
  {
    ($ParentStatus,$ParentUpdateTime) = NC::GetNodeStatus
($Parent);
    if( $ParentUpdateTime >= $NodeUpdateTime )
    {
      # Using TriggerFlag to store name of trigger to be fired. If
any
      # parent is found to be up, then the flag will be set to
down. If
      # all parents are down or unreachable, then the flag will be
set
      # to unreachable. If no parents are down and at least one
parent
      # is testing, set flag to testing. Otherwise, it will remain
not
      # set and we will update the node's current status and time.
Testing
      # handles the case where one parent is testing and another
is
      # unreachable. We need to make sure we do not mark the node
as
      # unreachable until the parent node in testing goes to some
final
      # state because that state could be agent down which is
treated
      # as up.
      if( ($ParentStatus eq "Down" || $ParentStatus eq
"UnReachable") && $TriggerFlag eq "NotSet")
      {
        $TriggerFlag = "UnReachable";
      }
      elsif( $ParentStatus eq "Up" )
      {
        $TriggerFlag = "Down";
      }
    }
  }
}
```

```
        elsif( $ParentStatus eq "Testing" && $TriggerFlag ne "Down"
)
    {
        $TriggerFlag = "Testing";
    }
else
    {
        # Remember that we have at least one parent that hasn't been
updated.
        $ParentNotUpdated = 1;
    }
}
else
{
    # If no parents, assume node is down.
    $TriggerFlag = "Down";
}

# If I have at least one parent not updated and I do not have
# any Up parents, Set TriggerFlag to testing.
if( $ParentNotUpdated && $TriggerFlag ne "Down" )
{
    $TriggerFlag = "Testing";
}

if( $TriggerFlag ne "Testing" )
{
    # Fire trigger if node's status should change.
    if( $TriggerFlag ne $LastNodeStatus )
    {
        # Fire trigger
        FireTrigger( $TriggerFlag );
    }
}
```

TestParentSetNode Perl Subroutine

For the DwnStrmSnmpStatus and DwnStrmIcmpStatus alarms, if an alarm is in a DeviceDown or Unreachable state, the ERROR trigger is fired every time the node is polled and doesn't respond. Each resulting ERROR transition calls the TestParentSetNode Perl subroutine.

The TestParentSetNode Perl subroutine tests the parent node status and determines the status of the node by doing the following:

- If the node has parents, TestParentSetNode evaluates each parent's last update time. Based on the following rules, TestParentSetNode sets a flag (TriggerFlag) that determines what trigger, if any, should be fired.
 - If no parents have an update time more recent than the node's update time, then TriggerFlag is set to Testing.
 - If at least one parent has a more recent update time but is not up, the flag is set to Testing.
 - If at least one parent has a more recent update time and is up, the flag is set to Down, regardless of the status or time of last update of any other parent.
 - If all parents have more recent update times and no parent is up or in testing, the flag is set to Unreachable.
- If the node has no parents, TriggerFlag is set to Down.

If TriggerFlag is set to Testing, TestParentSetNode does nothing because TestParentSetNode must have more information to make an accurate decision. If the alarm should be in another state, TestParentSetNode fires the appropriate trigger to transition the alarm into that state. If the alarm is already in the correct state, TestParentSetNode just refreshes the node update time so the node's children can accurately update their statuses based on the node's status.

The code for this subroutine follows:

```
# The purpose of this subroutine is to test the parent
# node status and, if the node is not in a terminal state
# but should be, fire a trigger to make it so. If the node
# is already in the correct state, just refresh the node
# update time. You must make sure that all parents are
# being monitored with the status alarms.
use NC;
my $NodeUpdateTime;    # Last time node status was updated
my $LastNodeStatus;   # Last node status
my @Parents = ();      # Array of parents
my $Parent;           # Parent Node
my $ParentUpdateTime; # Last time parent node status was updated
my $ParentStatus;     # Last parent status
my $TriggerFlag = "NotSet";
my $ParentNotUpdated = 0; # Remember if we have any parents not
updated
#Define all triggers that can be fired
DefineTrigger('UnReachable');
DefineTrigger('Down');
```

```
DefineTrigger('Testing');
# Get the last node status and update time for this node
($LastNodeStatus,$NodeUpdateTime) = NC::GetNodeStatus($NodeName);
# Get the array of parents for this node
@Parents = NC::GetParents($NodeName);
if( defined( $Parents[0] ) )
{
  # Test each parent, if any are ok, we assume the node
  # is reachable. Parents update time must be past the
  # last time the node was updated or we can't assume the
  # status is accurate.
  foreach $Parent (@Parents)
  {
    ($ParentStatus,$ParentUpdateTime) = NC::GetNodeStatus
($Parent);
    if( $ParentUpdateTime >= $NodeUpdateTime )
    {
      # Using TriggerFlag to store name of trigger to be fired. If
any
      # parent is found to be up, then the flag will be set to
down. If
      # all parents are down or unreachable, then the flag will be
set
      # to unreachable. If no parents are down and at least one
parent
      # is testing, set flag to testing. Otherwise, it will remain
not
      # set and we will update the node's current status and time.
Testing
      # handles the case where one parent is testing and another
is
      # unreachable. We need to make sure we do not mark the node
as
      # unreachable until the parent node in testing goes to some
final
      # state because that state could be agent down which is
treated
      # as up.
      if( ($ParentStatus eq "Down" || $ParentStatus eq
"UnReachable") && $TriggerFlag eq "NotSet" )
      {
        $TriggerFlag = "UnReachable";
      }
      elsif( $ParentStatus eq "Up" )
      {
        $TriggerFlag = "Down";
      }
    }
  }
}
```

```
        elseif( $ParentStatus eq "Testing" && $TriggerFlag ne "Down"
)
    {
        $TriggerFlag = "Testing";
    }
else
    {
        # Remember that we have at least one parent that hasn't been
updated.
        $ParentNotUpdated = 1;
    }
}
else
{
    # Node does not have parents so assume down
    $TriggerFlag = "Down";
}

# If I have at least one parent not updated and I do not have
# any up parents, Set TriggerFlag to testing.
if( $ParentNotUpdated && $TriggerFlag ne "Down" )
{
    $TriggerFlag = "Testing";
}
if( $TriggerFlag ne "Testing" )
{
    # Fire trigger if node's status should change. Otherwise
# refresh the time for the node's current state.
    if( $TriggerFlag ne $LastNodeStatus )
    {
        # Fire trigger
        FireTrigger( $TriggerFlag );
    }
else
    {
        # Refresh node status
        NC::SetNodeStatus( $NodeName, $LastNodeStatus );
    }
}
```


This appendix explains the error and information messages that you might encounter while using NerveCenter. Possible causes and solutions for the errors are included.

All NerveCenter error messages are written to the host syslog facility. To view messages in the syslog, read the file `/var/adm/messages` with a text editor or a command such as `more`.

Each error description is formatted in the following way:

```
Category error_message_number: message: [code_number]
```

Each message is assigned a category, which has a corresponding number. The line listed in the log uses a number to indicate a category, as follows:

Table 31: Error Message Categories

Number	Category
1	NC Server Manager
2	NC Alarm Manager
3	NC Trap Manager
4	NC Poll Manager
5	NC Action Manager
6	NC Protocol Manager
7	NC PA Resync Manager
8	NC Service
9	NC Inform NerveCenter Manager
11	NC LogToFile Manager
12	NC FlatFile Manager
13	NC Alarm Filter Manager
14	NC Deserialize Manager
16	NC DB Manager

The error message number indicates the error type. The error numbers are organized as follows:

Table 32: Error Message Numbers

Number Range	Type of Error
0-999	Users should call customer support.
1000-1999	User can resolve the problem.
2000-2999	User is warned of an event.
3000-3999	User is given an informational message.

The error messages are explained in the following sections:

- ["Alarm Filter Manager Error Messages" on the facing page](#)
- ["Deserialize Manager Error Messages" on the facing page](#)
- ["Flatfile Error Messages" on the facing page](#)
- ["Inform NerveCenter Error Messages" on the facing page](#)
- ["LogToFile Manager Error Messages" on page 414](#)
- ["Poll Manager Error Messages" on page 414](#)
- ["Protocol Manager Error Messages" on page 414](#)
- ["PA Resync Manager Error Messages" on page 415](#)
- ["Server Manager Error Messages" on page 417](#)
- ["Trap Manager Error Messages" on page 419](#)
- ["NerveCenter installation Error Messages" on page 420](#)

Alarm Filter Manager Error Messages

Table 33: Alarm Filter Manager Error Messages

#	Error	Resolution
1	Lookup failed on line number <i>value</i> in File <i>value</i> .	
3001	Alarm Filter Manager Initialization successfully finished	

Deserialize Manager Error Messages

Table 34: Deserialize Manager Error Messages

#	Error	Resolution
1	Lookup failed on line number <i>value</i> in File <i>value</i> .	
3001	Deserialize Thread Manager Initialization successfully finished	

Flatfile Error Messages

Table 35: Flatfile Manager Error Messages

#	Error	Resolution
1	Lookup failed on line number <i>value</i> in File <i>value</i> .	
3001	Flat File Initialization successfully finished	

Inform NerveCenter Error Messages

Table 36: Inform NerveCenter Manager Error Messages

#	Error	Resolution
1	Lookup failed on line number <i>value</i> in File <i>value</i> .	
3001	InformNC Manager Initialization successfully finished	

LogToFile Manager Error Messages

Table 37: Log to File Manager Error Messages

#	Error	Resolution
1	Lookup failed on line number <i>value</i> in File <i>value</i> .	
3001	LogToFile Manager Initialization successfully finished	

Poll Manager Error Messages

Table 38: Poll Manager Error Messages

#	Error
3001	Poll Manager Initialization successfully finished
3002	CPollManagerWnd:OnPollOnOff, PreCompild of PollEvent with Poll Id %ld failed

Protocol Manager Error Messages

Table 39: Protocol Manager Error Messages

#	Error	Resolution
1	Building copy of node list failed.	N/A
2	Building copy of poll property list failed.	N/A
3	I.nitialization of protocol methods failed	N/A
4	Initialization of ping socket failed.	N/A
5	Creation of SNMP socket failed, socket error code: %d	N/A
6	Error in ping socket: %s	N/A
7	Error in ping socket: create socket failed.	N/A
8	Error in ping socket: async select failed.	N/A
1000	Looking for the %s key in the configuration settings.	Use the Administrator to enter the SNMP values in the configuration settings.
1001	Ncuser user ID is not found.	Add ncuser user ID to your system.
3000	Initialization successfully finished.	N/A
3001	Invalid value in configuration settings for SNMP retry interval, using default of 10 seconds.	Use the Administrator to enter an SNMP retry interval.

#	Error	Resolution
3002	Invalid value in configuration settings for number of SNMP retries, using default of 3 retries.	Use the Administrator to enter a number of SNMP retries.
3003	Invalid value in configuration settings for default SNMP port, using default of 161.	Use the Administrator to enter the default SNMP port number.

PA Resync Manager Error Messages

Table 40: PA Resync Manager Error Messages

#	Error	Resolution
1	Error getting local host name for encoding resync request, socket error code: %d	N/A
2	Encoding resync request failed	N/A
3	Sending resync request failed with zero bytes sent	N/A
4	Sending resync request failed: %s	N/A
5	Memory allocation error, trying to notify of connection status	N/A
6	Memory allocation error, creating node list	N/A
7	Memory allocation error, creating a resync node	N/A
8	Parent status not sent during resync	
10	Parents not computed during resync with map host. Check OVPA. OVPA database must have nc host node.	
500	Socket Error: (%d)	
1000	Error looking for the %s key in the NerveCenter configuration settings	Use the Administrator to enter configuration settings.
1001	Attempt to connect to %s on port %d failed: %s	Make sure the platform host is up and running and that the name exists in the hosts file.

#	Error	Resolution
1002	Resync connection attempt failed: %d	Make sure the platform host is up and the platform adapter is running.
1500	The connection to % was closed	
1501	Send failed with zero bytes sent	
1505	%s. The address is already in use	Make sure you are not running two instances of the same application on the same machine.
1506	%s. The connection was aborted due to timeout or other failure	Make sure the physical network connections are present.
1507	%s. The attempt to connect was refused	Make sure the server is running on the remote host.
1508	%s. The connection was reset by the remote side	Make sure the remote peer is up and running.
1509	%s. A destination address is required	A destination address or host name is required.
1510	%s. The remote host cannot be reached	Make sure the routers are working properly.
1511	%s. Too many open files	Close any open files.
1512	%s. The network subsystem is down	Reboot the machine.
1513	%s. The network dropped the connection	Make sure the peer is running and the network connections are working.
1514	%s. No buffer space is available	This might be because you are running several applications, or an application is not releasing resources.
1515	%s. The network cannot be reached from this host at this time	Make sure the routers are functioning properly.
1516	%s. Attempt to connect timed out without establishing a connection	Make sure the machine is running and on the network.
1517	%s. The host cannot be found	Make sure you can ping the host, check you hosts file or DNS server.
1518	The network subsystem is unavailable	Make sure the network services are started on machine.

#	Error	Resolution
1519	%s. Invalid host name specified for destination	The host name cannot be resolved to an IP address. Enter the name to the hosts file or DNS server.
1520	The specified address in not available	Make sure the host name is not zero. Try pinging the host.
3000	initialization successfully finished	N/A
3001	Node resync from map host was not requested because either host name or port number is missing	If you are trying to disable a connection to the platform adapter, then this message is OK. If you want to be connected to the platform adapter, then use the Administrator to check the map host settings.
3500	Connection to %s was successful	N/A

Server Manager Error Messages

Table 41: Server Manager Error Messages

#	Error	Resolution
2	Perl create failed.	N/A
3	Initialization of <i>value</i> manager thread failed.	N/A
4	Failed to restore MibDirectory in configuration settings.	N/A
5	Failed to open configuration settings while trying to restore mib information.	N/A
6	Discrepancy in data. File: SERVER_CS.CPP, Line: <i>value</i> .	N/A
10	Conflict in data. File: SERVER_CS.CPP, Line: <i>value</i> .	N/A
11	Internal Error. File: SERVER_CS.CPP, Line: <i>value</i> .	N/A
20	Cannot read configuration settings value: Bind.	N/A
21	Cannot connect to Tcpip configuration settings information.	N/A
22	Cannot read configuration settings value: IPAddress.	N/A

#	Error	Resolution
23	Couldn't find <i>value</i> in map.	N/A
24	Error while reading database. Poll/Mask: <i>value</i> uses a simple trigger that doesn't exist in database.	N/A
25	Please report error number <i>value</i> to technical support.	N/A
26	User validation failed: Unable to communicate with ncsecurity process : <i>value</i> .	~
1002	Initialization failed, cannot find nperl.pl.	Check NCPperl.pl location.
1003	Failed to open MIB: <i>value</i> .	Check MIB location.
1004	Failed to parse MIB.	Invalid MIB. Check configuration to see if the correct MIB is specified.
1010	Failed to validate poll: <i>value</i> . The poll will be turned off.	Check the poll condition using the Client Application.
1100	<i>value</i> (database error).	Try to resolve using the message. If not, call support.
1103	Version table validation failed. NC_Version table doesn't exist in database.	Upgrade the NerveCenter database.
1200	Failed to open configuration settings while trying to restore mib information.	Use the NerveCenter Administrator to check the configuration settings. Invalid key is likely.
1202	Cannot connect to configuration settings.	Use the NerveCenter Administrator to check the configuration settings. Invalid key is likely.
1203	Cannot open key <i>value</i> .	Use the NerveCenter Administrator to check the configuration settings.
1204	Cannot add value <i>value</i> .	Use the NerveCenter Administrator to check the configuration settings. Invalid key is likely.
1205	Cannot read configuration settings value in MapSubNets key.	Use the NerveCenter Administrator to check the configuration settings. Invalid key is likely.
1206	Invalid configuration settings Entry for the value Method in the Platform key.	Only Manual and Auto are allowed. Check for case.
1207	Cannot read configuration settings value: <i>value</i>	Use the NerveCenter Administrator to check the configuration settings. Invalid key is likely.

#	Error	Resolution
1208	Cannot write configuration settings Value: <i>value</i>	Use the NerveCenter Administrator to check the configuration settings. Invalid key is likely.
1300	<i>value</i> (Import behavior/database error).	Try to resolve using the message. If not, call - support.
1313	Server alarm instance maximum exceeded. Please restart Server.	Restart server.
3001	Request to delete the node <i>value</i> failed because the node doesn't exist.	N/A
3002	Failed to find socket in server's map. Line: <i>value</i> .	
3003	Exiting due to a SIGTERM signal.	
3004	Primary thread initialization successful.	

Trap Manager Error Messages

Table 42: Trap Manager Error Messages

#	Error	Resolution
1	Error in TrapManagerWnd::Initialize - failed to create GetHostByAddr thread.	
2	Error in TrapManagerWnd::LaunchTrapper - failed to create trapper process.	
3	Error in TrapManagerWnd::CreateCheckTrapperThread - failed to create new thread.	
8	Error in TrapManagerWnd::Initialize - Failed to create trap stream socket.	
9	Error in TrapManagerWnd::Initialize - Failed to listen on trap stream socket.	
10	Error in TrapManagerWnd::OnTraceTraps - Failed to create trace file for traps.	
1001	CTrapManagerWnd::OnTrapExist - gethostbyname from trap data with snmptrap failed for <i>value</i> .	
1003	CTrapManagerWnd::OnInvalidSignature - Error in receiving data on NC socket.	Check for consistent version numbers of trapper and NerveCenter executables.

#	Error	Resolution
2001	MS Trap service threw exception in GetTrap.	Make sure you aren't making SNMP get requests to port 162.
2002	Error processing trap data.	Make sure you aren't making SNMP get requests to port 162.
3001	Trap Manager Initialization successfully finished.	
3002	Check Trapper—Trapper process died. restarting Trapper.	

NerveCenter installation Error Messages

Table 43: NerveCenter Installation Error Messages (UNIX)

Error	Resolution
Space under <i>dirname</i> is INSUFFICIENT to install LogMatrix NerveCenter	Free up space in the file system by removing files, or choose another place for installation.
The directory <i>dirname</i> must reside on a local disk	The directory you specified for NerveCenter installation is on a disk that is not on the local file system. Pick a new directory or re-mount the disk.
Write permission is required by root for <i>dirname</i> directory	The directory you specified for NerveCenter installation does not have write permission for root. Choose another directory or change the permissions.
Please create the desired destination directory for NerveCenter and re-run the installation script	The directory you specified for NerveCenter installation does not exist. Choose another directory or create the original.
Invalid mount point	The installation script could not find the CD-ROM drive and prompted you for its location. The path you specified was not valid. Verify that the drive exists, is mounted, and is configured correctly.
<i>ProcessName</i> is running on the system. Please exit from (or kill) <i>processName</i> process.	The installation script found that the <i>nervectr</i> or <i>ovw</i> process was running. Exit from or kill the process and re-run the installation script.
These processes must be stopped before NerveCenter can be installed. Please kill these processes and re-run the installation script.	The installation script found processes that need to be killed before installation, you were prompted to stop them, and you said no. You must manually exit from or kill the processes and re-run the installation script.

Error	Resolution
<i>hostname</i> is not a valid host name	The host that you provided to the script is not a valid host. Check the name of the host (capitalization, spelling, and so on) and try again.
I don't know how to install on this architecture	Installation is supported for Solaris. The script issues this message if attempting to install on an architecture that is not in this set.
Can't cd to <i>installation_path</i> /userfiles	Make sure the directory exists and has appropriate permissions.
Can't open <i>hostname.conf</i>	The script couldn't create the file or couldn't open an existing configuration file. Check <i>installation_path</i> /userfiles to make sure that root has permission to write in this directory, that <i>hostname.conf</i> has read permission set, if it exists, and that <i>localhost.conf</i> exists and has read permission set.
Can't create <i>hostname.ncdb</i> Can't create <i>hostname.node</i>	The script was attempting to create the indicated file by copying data from another file. Check <i>installation_path</i> /userfiles to make sure that root has permission to write in this directory, and that <i>localhost.ext</i> exists and has read permission set.
Can't open /etc/rc Couldn't re-create /etc/rc Couldn't modify /etc/rc	The script couldn't modify /etc/rc to call the NerveCenter rc script. Edit the file and add a line that executes <i>installation_path</i> /bin/rc.openservice. There's no need to rerun the installation script after this correction.
Can't append to /etc/rc.local	The script couldn't modify /etc/rc.local to call the NerveCenter rc script. Edit the file and add a line that executes <i>installation_path</i> /bin/rc.openservice. There's no need to rerun the installation script after this correction.
Can't create /etc/rc2.d/K94ncservice on Solaris	The script couldn't create the NerveCenter rc script /etc/rc2.d/K94ncservice on Solaris. Copy <i>installation_path</i> /bin/rc.openservice to /etc/rc2.d/K94ncservice. There's no need to rerun the installation script after this correction.

Error	Resolution
<p>An error occurred in trying to contact the Server "<i>hostname</i>". As a result, the information that you have specified cannot be used to complete this NIS update.</p> <p>Unable to modify <i>filename</i>. It doesn't exist!</p> <p>Unable to modify <i>filename</i>. File size is 0!</p>	<p>The script was attempting to update system services and failed. Correct the specific error (perhaps the host name or file name was entered incorrectly) and rerun the script. If the error isn't easily corrected, you can edit <code>/etc/services</code> yourself. Make sure that the following lines are included in the file:</p> <pre>SNMP 161/udp SNMP-trap 162/udp</pre> <p>If you're running NIS, be sure to make these changes on the NIS server, change to the NIS directory, and run <code>make services</code>.</p>



- A -

- Action Router
 - alarm action 269
 - creating a rule 318
- Action Router alarm action 15, 269
- Action Router Rule Definition window 317, 319
- Action Router Rule List window 316
- Action Router rules
 - conditions, creating 318
 - Counter() 299
 - deleting 337
 - functions 321
 - In() 179
 - listing, existing 316
 - rule actions, defining 323
 - variables, NerveCenter 301
- Action Router tool 15, 269
- adding nodes
 - IP Lookup 95
- Administrator, NerveCenter 19
- alarm-instance filters 62
- alarm action 249
 - associating with a transition 249
- alarm actions 12, 18, 29, 269
 - Action Router 15, 269
 - Alarm Counter 271
 - Beep 274
 - Clear Trigger 275
 - Command 277, 301
 - Delete Node 278
 - EventLog 279
 - Fire Trigger 280
 - FireTrigger 14, 280
 - Forward Trap 284
 - Inform NerveCenter 286
 - Inform
 - Netcool/OMNIBus 289
 - Inform OpenView 286
 - Inform Platform 289
 - Inform variable bindings 234
 - Log to File 291
 - logging-Event Log 279
 - logging-Log to File 291
 - mailing-SMTP mail 311
 - Notes 292
 - Perl Subroutine 294-295, 301
 - send SMS 304
 - Send SMS 304
 - Send Trap 305, 308
 - Set Attribute 155, 308
 - SMTP Mail 311
 - SNMP Set 312-313
- Alarm Counter action 271
- Alarm Counter Action dialog 273
- Alarm Counter alarm action 271
- alarm definition
 - associating actions 249
 - defining a state 247, 249
 - severity 346
- Alarm Definition List window 86, 90, 238, 260, 354, 356
- Alarm Definition window 91, 239, 242, 260
- Alarm Filter error messages 413
- alarm scope 42, 244
- alarm severity
 - filtering 70
- Alarm Summary window
 - creating 241
 - defining 247, 249
 - entering notes 254
 - exporting behavior models 353, 357
 - filtering 66, 68
 - severity 346
- alarms 17, 27
 - correlation expressions 261
 - defining 241
 - deleting 337
 - documenting 254
 - enabling 259
 - examples 29, 50
 - filtering by IP range 63
 - filtering rules 79
 - IF-IfFramePVC 330
 - IF-IfStatus 327
 - IfLoad state diagram 51
 - interface-type 329
 - IPSweep, enabling 90
 - listing 238
 - monitoring loads 29
 - notes 254
 - performing actions conditionally 315
 - property groups, changing 338
 - scope, changing 339
 - state diagrams 29, 50
 - TcpRetransMon 50
 - using 237
- assigning
 - property groups to nodes 133, 143-145, 147, 155, 158

AssignPropertyGroup()
 function 133, 147, 175

associating
 actions with transitions 251

attributes
 nodes, changing 342
 polls 37
 severities used by
 NerveCenter 345
 severities 344
 trap masks 40
 triggers 39
 variable bindings 205

authentication 102, 105

authentication password 102,
 105

Authentication Protocol for
 SNMP v3 Nodes 105

auto-classification 99, 107, 114-
 115

- B -

base objects 205

Beep Action dialog 274

Beep alarm action 274

behavior model
 exporting 353, 357
 importing 364-365
 IPSweep 85-86

behavior models 6, 17, 47
 creating 47
 creating multi-alarm 325
 definition 27
 design 27
 diagram 47
 Discovery 85
 example 50
 exporting 357
 exporting to files 355
 exporting to other

servers 353

files 355, 357, 364

importing 364

IPSweep 84, 90

multi-alarm 281, 325

predefined 18, 27

boolean expression, creating
 alarm from 261

Boolean functions 261

built-in triggers 220, 225

CANNOT_SEND 223, 225

ERROR 220-223, 225

example 228

firing sequence 223

how NerveCenter fires 220

ICMP errors 221, 223-224

ICMP_ERROR 221-223,
 225

ICMP_TIMEOUT 221, 223,
 225

ICMP_UNKNOWN_
 ERROR 225

INFORM_CONNECTION_
 DOWN 223

INFORM_CONNECTION_
 UP 223

INFORMS_LOST 223

list 225

list of 225

matching errors 224

NET_UNREACHABLE 221-
 222, 225

NODE_
 UNREACHABLE
 221-222, 226

order fired 223

PORT_
 UNREACHABLE
 221-222, 226

RESPONSE 223, 226

SNMP_
 AUTHORIZER

R 226

SNMP_BADVALUE 226

SNMP_DECRYPTION_
 ERROR 226

SNMP_ENDOFTABLE 226

SNMP_ERROR 220, 223

SNMP_GENERR 226

SNMP_
 NOSUCHNAME 226

SNMP_NOT_IN_TIME_
 WINDOW 227

SNMP_READONLY 227

SNMP_TIMEOUT 220-221,
 223, 227

SNMP_TOOBIG 227

SNMP_UNAVAILABLE_
 CONTEXT 227

SNMP_UNKNOWN_
 CONTEXT 227

SNMP_UNKNOWN_
 ENGINEID 227

SNMP_UNKNOWN_
 USERNAME 227

SNMP_UNSUPPORTED_
 SEC_LEVEL 227

SNMP_WRONG_
 DIGEST 227

UNKNOWN_ERROR 227

- C -

CANNOT_SEND built-in
 trigger 223, 225

CaseContainsString() 182

CaseContainsWord() 182

categorizing nodes 160

changing
 alarm property 338
 alarm scope 339
 node attributes 342
 node property group 339
 object property 338

- object property group 338
 - poll property 338
 - properties 338
 - property groups 339
 - state icons size 248
 - transition icon sizes 252
 - classification 107
 - classifying SNMP version for all nodes 110
 - classifying SNMP version for one or more nodes 109
 - confirming the SNMP version for a node 111
 - how NerveCenter classifies nodes 115
 - when NerveCenter classifies nodes 114
 - classification of SNMP version 99, 107
 - all nodes manually 110
 - confirming the version of a node 111
 - one or more nodes manually 109
 - Clear Trigger Action dialog 276
 - Clear Trigger alarm action 275
 - CLI 22
 - client
 - changing server port 61
 - Client
 - configuring IPSweep 86
 - Client, NerveCenter 21
 - Code (ICMP field) 221-222
 - colors, creating custom 350
 - Command Action dialog 89, 277
 - Command alarm action 277
 - variables 301
 - variables, NerveCenter 301
 - command line interface 22
 - conditional alarm actions 269, 315
 - conditions
 - Action Router rules 318
 - finding sequences 10
 - finding set of network 9
 - network, detecting 6, 28
 - persistent network 8
 - responding to network 12
 - conditions, tracking network 28
 - configuring SNMP v3 nodes 99, 102, 105, 111
 - ContainsString() 182
 - ContainsWord() 182
 - context 102, 105
 - copying
 - objects 334-335
 - property groups 335
 - corrective actions 14
 - correlating conditions 7
 - correlation expression list window 262, 265
 - correlation expression notes window 268
 - Correlation expression overview 261
 - correlation expressions
 - about 261
 - copying 265
 - creating 262
 - creating alarm from 266
 - notes 268
 - Counter() function 299
 - create alarm using correlation expression window 266
 - creating
 - Action Router rule conditions 318
 - behavior models 47
 - colors, custom 350
 - multi-alarm behavior models 325
 - poll conditions 166-167
 - properties 132
 - property groups, based on existing 134
 - property groups, based on MIBs 135
 - property groups, manually 138
 - severities 348
 - trap masks 230
 - trigger functions 202
- D -
- data sets
 - nodes 30
 - polls 37
 - severities 344
 - trap masks 39
 - triggers 39
 - data sources, other 219
 - Database Wizard
 - populating the node data table 86
 - database, NerveCenter 16
 - default severities 346
 - DefineTrigger() function 175
 - DefineTrigger() function 175
 - defining
 - Action Router rule conditions 319
 - alarms 241
 - node sets 6
 - nodes 84, 99, 117
 - nodes, manually 92

- Perl subroutines 295
- polls 164
- properties 129
- property groups 129
- rule actions 323
- states 245, 247
- transitions 249-250
- trap masks 198
- Delete Node alarm action 278
- deleting
 - Action Router rules 337
 - alarms 337
 - nodes 337
 - objects 336
 - OID to property group mappings 336
 - OpC masks 337
 - Perl subroutines 337
 - polls 337
 - property groups 336
 - severities 336
 - states 249
 - transitions 254
 - trap masks 337
- delta() 170
- Deserialize Manager error messages 413
- Destination Address (ICMP field) 221, 224
- detecting condition persistence 8
- detecting conditions 6, 28
- digest keys 102, 105, 119
- disconnecting 83
- discovering nodes 84-85, 99, 117
 - customize IPSweep 86
- Discovery behavior model 85
- discovery of SNMP v3 agents configuring NerveCenter 86
- documentation
 - conventions 3
 - feedback 3
- documenting
 - alarms 254
 - Perl subroutines 296
 - polls 186-187, 190
- downstream alarm suppression 300
- DumpParentsToFile() 300
- E -**
- edit correlation expression window 263, 265
- elapsed (poll condition function) 170
- enabling
 - alarms 259
 - IPSweep alarm 90
 - objects 333
 - polls 191
 - trap masks 216
- end_poll (poll condition function) 171
- ERROR built-in trigger 220-223, 225
- error messages 220, 225
 - Alarm Filter 413
 - Deserialize Manager 413
 - Flatfile Manager 413
 - ICMP 221, 223-224
 - Inform Product Manager 413
 - LogToFile Manager 414
 - PA Resync Manager 415
 - Poll Manager 414
 - Protocol Manager 414
 - Server Manager 417
 - SNMP 220, 223-224
 - Trap Manager 419
 - UNIX installations 420
- user interface 411
- error status
 - SNMP v3 operations 124
- Event Log Action dialog 279
- Event Log alarm action 279
- EventLog alarm action 279
- Expanded Color window 350
- Expanded Rule Condition page 322
- Export Model/Object dialog 354, 356
- Export Objects and Nodes dialog 358, 361
- exporting 353
 - behavior models 353, 357
 - behavior models, to files 355
 - behavior models, to other servers 353
 - individual NerveCenter objects 357, 360, 362
 - node relationships to files 300
 - nodes 353
 - nodes to a file 360
 - nodes to other servers 357
 - objects 353, 362
 - objects to a file 360
 - objects to other servers 357
 - relationships node 300
- F -**
- filter 63, 66, 68
 - associating with server 77
- filtering alarms
 - by IP range 63
 - by property group 74
 - by severity 70
 - IP range 66, 68
- finding set of network conditions 9

- Fire Trigger Action dialog 88, 281
 - Fire Trigger alarm action 280
 - FireTrigger alarm action 14
 - FireTrigger() function 177
 - FireTrigger() function 177
 - Flatfile Manager error messages 413
 - Forward Trap Action dialog 285
 - Forward Trap alarm action 284
 - functions 297
 - Action Router rules 321
 - AssignPropertyGroup 175
 - AssignPropertyGroup() 133, 147
 - CaseContainsString() 182
 - CaseContainsWord() 182
 - ContainsString() 182
 - ContainsWord() 182
 - Counter() 299
 - DefineTrigger 175
 - DefineTrigger() 175
 - DumpParentsToFile() 300
 - Fire Trigger 177
 - FireTrigger() 177
 - In 179
 - In() 179
 - LoadParentsFromFile() 300
 - node relationship functions 300
 - Perl subroutines 297
 - poll conditions 168
 - poll conditions, for 170
 - RemoveAllParents() 300
 - string matching 181
 - triggers 203
 - used in Perl subroutines 297
 - used in poll conditions 168
 - used in trap mask trigger functions 203
 - variable bindings 204
- G -**
- getbulk 108
 - GetRequest 220, 224
- H -**
- heartbeat messaging 80
 - deactivating 82
 - interval 81
- I -**
- ICMP error messages 221, 223-224
 - ICMP fields 221-222, 224
 - ICMP requests 221, 224
 - ICMP_ERROR built-in trigger 221-223, 225
 - ICMP_TIMEOUT built-in trigger 221, 223, 225
 - ICMP_UNKNOWN_ERROR built-in trigger 225
 - IF-IfColdWarmStart alarm 331
 - IF-IfFramePVC alarm 330
 - IF-IfNmDemand alarm 332
 - IF-IfStatus alarm 327
 - IF-SelectType Perl subroutine 328
 - IfUpDownStatusByType behavior model 325
 - Import Behavior Model dialog 365
 - importing 364
 - behavior model files 364
 - behavior models 364
 - node files 364
 - node relationships from files 300
 - nodes 353
 - object files 364
 - objects 353
 - relationships node 300
 - using ImportUtil 365
 - ImportUtil.exe 365
 - In() function 179
 - in() function 179
 - Inform 290, 346
 - Inform Action dialog 288-289
 - Inform alarm action 286, 289
 - trap variable bindings 234
 - variable bindings 234
 - Inform Platform alarm action 289
 - Inform Product Manager error messages 413
 - INFORM_CONNECTION_DOWN built-in trigger 223
 - INFORM_CONNECTION_UP built-in trigger 223
 - INFORMS_LOST built-in trigger 223
 - instances 205
 - integration with network management platforms 24
 - interface-type alarms 329
 - IP fields 224
 - IP filters 63
 - examples 68
 - subnet filter rules 66
 - IP Lookup 95
 - IPSweep 85-86
 - IPSweep alarm definition 87

enabling 90
 modifying 86
 state diagram 87
 IPSweep behavior model 84
 ipsweep.exe 63

- J -

jump_to (poll condition
 function) 171

- K -

keys 102, 105
 digest 119

- L -

levels of severities 345
 listing
 Action Router rules,
 existing 316
 alarms 238
 polls 162
 properties 129, 131
 property groups 129-130
 trap masks 193, 196
 LoadParentsFromFile() 300
 loads alarm, monitoring
 interface 29
 Log to Database alarm action
 variables, NerveCenter 301
 Log to File Action dialog 291
 Log to File alarm action 291
 variables, NerveCenter 301
 logging 13
 logging alarm data
 Event Log 279
 Log to File 291
 logs
 SNMP v3 operations 120

LogToFile Manager error
 messages 414
 looking for high traffic on four
 interfaces 43

- M -

mail alarm actions
 sending-SMTP mail 311
 sending SMS 304
 SMS 304
 SMTP mail 311
 main NerveCenter
 components 15
 managing NerveCenter
 objects 333
 mapping OIDs to property
 groups 158
 mask definition window 217,
 231
 Mask Definition window 199
 Mask List window 196, 198,
 216, 230
 menus 182
 Merge or Overwrite Property
 Group window 137
 MIB
 sharing information 59
 MIB base objects 312
 MIB objects 160
 MIB to property group
 window 136
 mod files 355, 357, 360-361, 364
 modifying
 IPSweep alarm 86
 monitoring
 interface loads alarm 29
 nodes, a set of 29
 viewing alarm notes 254
 multi-alarm behavior
 models 281, 325
 multi-homed nodes 224

multiple NerveCenter
 servers 229

- N -

NC AlarmCounters 179
 NCContext 102, 105
 NCUser 102, 105
 NerveCenter
 Action Router tool 15
 Administrator 19
 Client 21
 data sources, other 219
 database 16
 functions for poll
 conditions 170
 node management 6
 objects 30
 Server 16
 servers, multiple 24, 229
 severities 343
 NerveCenter built-in
 triggers 220, 225
 ICMP errors 221, 223-224
 NerveCenter Client
 starting 53
 NerveCenter installation error
 messages (UNIX) 420
 NerveCenter user security for
 SNMP v3 102, 105
 NerveCenter variables 301
 NET_UNREACHABLE built-in
 trigger 221-222, 225
 network conditions
 detecting 6, 28
 finding set of 9
 persistent 8
 responding to 12
 tracking 28
 network management platform
 filtering by IP subnet 63, 66,
 68

- map colors 345
- sending Informs 346
- network management
 - strategy 22
- New Severity window 348
- node classification 99, 107, 114-115
 - all nodes manually 110
 - confirming the SNMP version of a node 111
 - one or more nodes manually 109
- Node Definition window 93, 95, 144-145
- Node List window 92, 95, 100, 103, 105, 109, 111, 144-146, 340, 342
- node relationship functions 300
- node source - server status
 - filtering by subnet 63, 66, 68
 - populating the database 63, 66, 68, 86
- Node status behavior models
 - adding to the database 85, 92
 - assigning property groups 143, 158, 175
 - SNMP classification 107, 109-111, 114-115
 - SNMP settings 99, 102, 105
 - SNMP Test Version poll 111
- NODE_UNREACHABLE built-in trigger 221-222, 226
- nodes 28, 30
 - assigning to property groups 133, 143-145, 147, 155, 158
 - changing attributes 342
 - data set 30
 - defining sets 6

- defining, manually 92
- deleting 337
- discovering 84, 99, 117
- exporting relationships to files 300
- exporting to a file 360
- exporting to other servers 357
- ICMP attributes 32-33
- importing 364
- importing relationships from files 300
- managing 6
- monitoring a set of 29
- multi-homed 224
- property groups, changing 339
- relationship with poll 49
- relationship with properties 48
- relationship with property groups 48
- relationships, exporting 300
- relationships, importing 300
- relationships, removing from database 300
- SNMP attribute 30
- source 84
- suppressing 340
- not_present (poll condition function) 172
- notes 186, 210, 254, 292
 - alarms 254
 - Perl subroutines 296
 - polls 186-187, 190
- Notes alarm action 292
- notification 13

- O -

- objects
 - copying 334-335
 - deleting 336
 - enabling 333
 - exporting to a file 360
 - exporting to other servers 357
 - files 360-361, 364
 - importing 364
 - NerveCenter 30
 - properties, changing 338
 - property groups, changing 338
 - types you can export 362
- objects, database 16
- OID to Property Group dialog 159
- OID to property group mappings 158
 - deleting 336
- OnResponse page 167
- OpC masks
 - deleting 337
- operations log 120
 - signing for errors 121
 - viewing 123
- overview of NerveCenter SNMP v3 support 99, 102, 105, 111

- P -

- PA Resync Manager error messages 415
- parent child relationships, nodes 300
- password 102, 105

- passwords
 - SNMP v3 119
- Perl
 - built-in triggers, use with 221
 - Counter() 299
 - defining subroutines 295
 - deleting subroutines 337
 - documenting 296
 - example 303
 - functions 297
 - In() 179
 - notes 296
 - pop-up menu 182
 - subroutines 151
 - variables, NerveCenter 301
- Perl functions
 - AssignPropertyGroup 175
 - DefineTrigger 175
 - Fire Trigger 177
 - In 179
 - string matching 181
- Perl subroutine
 - creating 295
 - defining 295
 - executing as an alarm
 - action 294
 - functions 297
 - variables 301
- Perl Subroutine Action
 - dialog 150, 154, 294
- Perl Subroutine alarm action 294
- Perl Subroutine Definition
 - window 296
- Perl Subroutine List window 295
- Perl subroutines
 - built-in triggers 221
 - IF-SelectType 328
- ping requests 221, 224
- platform names, associated with
 - severities 346
- poll condition 166
 - examples 184
 - functions used 168
- poll condition functions 170
 - delta() 170
 - elapsed 170
 - end_poll 171
 - jump_to 171
 - not_present 172
 - present 173
 - previous 173
 - start_at 174
- Poll Condition page 148
- poll conditions 147
 - creating 166-167
 - DefineTrigger() 175
 - examples 184-186
 - FireTrigger() 177
 - functions 168, 170
 - In() 179
 - variables, NerveCenter 301
- poll definition 164
 - copying 335
 - creating 164
 - defining 164, 166, 168, 184
 - deleting 337
 - entering notes 186
 - overriding node
 - suppression 341
 - poll condition 166, 168, 184
 - SNMP Test Version poll 111
 - suppressing or
 - unsuppressing 341
- Poll Definition window 163-164, 188, 192
- Poll List window 162, 164, 187, 191, 333, 341
- Poll Manager error
 - messages 414
- Poll Notes and Associations
 - dialog 189
- Poll pop-up menu 334
- polling
 - getbulk 108
- polls 27, 37
 - attributes 37
 - built-in triggers 220
 - conditions, creating 167
 - data set 37
 - defining 164
 - deleting 337
 - documenting 186-187, 190
 - enabling 191
 - listing 162
 - notes 186-187, 190
 - pending list 220
 - ping requests 221, 224
 - property groups,
 - changing 338
 - relationship with nodes 49
 - SNMP requests 220, 224
 - suppressible, making 340-341
 - using 161
- pop-up menu for Perl 182
- PORT_UNREACHABLE built-in
 - trigger 221-222, 226
- predefined behavior models 18, 27
- predefined NerveCenter
 - severities 346
- present (poll condition
 - function) 173
- previous (poll condition
 - function) 173
- privacy 102, 105
- privacy password 102, 105
- properties 6, 28, 36
 - changing 338
 - creating 132
 - defining 129
 - filtering 139

- listing 131
 - relationship with nodes 48
 - Property dialog 338
 - property filter
 - about 139
 - creating 141
 - deleting 143
 - editing 143
 - property group
 - assigning to nodes 143, 158, 175
 - creating 133-135
 - Property Group dialog 146, 339
 - Property Group List
 - window 134-135, 138, 141, 143, 335
 - property groups 28, 36
 - assigning to nodes 133, 143-145, 147, 155, 158
 - changing 338-339
 - copying 335
 - creating manually 138
 - creating, based on existing 134
 - creating, based on MIBs 135
 - defining 129
 - deleting 336
 - filtering alarms 74
 - listing 130
 - property groups and properties 36
 - property groups relationship with nodes 48
 - Protocol Manager error messages 414
- R -**
- RemoveAllParents() 300
 - responding to network conditions 12
 - RESPONSE built-in trigger 223, 226
 - routing alarm actions 269
 - Rule Action page 323
 - rule actions, defining 323
 - rules for alarm filters 79
- S -**
- scope 42, 244
 - changing 339
 - Scope dialog 340
 - scripts See Perl 269
 - security 102, 105
 - SNMP v3 118
 - security for SNMP v3 102, 105
 - Security Level for SNMP v3 Nodes 102
 - Send SMS alarm action 304
 - Send Trap Action dialog 306
 - Send Trap alarm action 305, 308
 - Sequence Number (IP field) 224
 - server
 - associating filters 77
 - automatic connection 57
 - changing port on client 61
 - connecting 54
 - deleting 60
 - disconnecting from 83
 - manual connection 55
 - selecting active 60
 - sharing MIB information 59
 - Server Manager error messages 417
 - Server Selection dialog 354, 359
 - server status
 - exporting behavior models to 353
 - servers
 - alarm filtering rules 79
 - multiple 229
 - servers, multiple NerveCenter 24
 - Set Attribute Action dialog 156, 310
 - Set Attribute alarm action 155, 308
 - severities 17, 344, 346
 - attributes 344-345
 - attributes used by NerveCenter 345
 - creating 348
 - data set 344
 - default 346
 - deleting 336
 - levels 345
 - map colors in NMPs 345
 - platform names 346
 - Severity List window 348
 - smart polling 6
 - SMTP mail action dialog 304, 311
 - SMTP mail alarm action 311
 - SMTP Mail alarm action 311
 - SNMP
 - getbulk 108
 - SNMP error messages 220, 223-224
 - SNMP errors 220, 223-224
 - SNMP requests 220, 224
 - SNMP Set Action window 312
 - SNMP Set alarm action 312
 - variable bindings 313

- SNMP settings 117
 - node classification 114-115
 - SNMP v3
 - built-in triggers 226-227
 - Changing Authentication Protocol 105
 - Changing Security Level 102
 - classification 107, 109-111, 114-115
 - digest keys 119
 - error status 124
 - node settings 99, 102, 105
 - operations log 120
 - passwords 119
 - security 102, 105, 118
 - security levels 118
 - security support 118
 - test poll 111
 - SNMP v3 support 117
 - node classification 99, 107, 114-115
 - test poll 126
 - SNMP_AUTHORIZATIONERR built-in trigger 226
 - SNMP_BADVALUE built-in trigger 226
 - SNMP_DECRYPTION_ERROR built-in trigger 226
 - SNMP_ENDOFTABLE built-in trigger 226
 - SNMP_ERROR built-in trigger 220, 223
 - SNMP_GENERR built-in trigger 226
 - SNMP_NOSUCHNAME built-in trigger 226
 - SNMP_NOT_IN_TIME_WINDOW built-in trigger 227
 - SNMP_READONLY built-in trigger 227
 - SNMP_TIMEOUT built-in trigger 220-221, 223, 227
 - SNMP_TOOBIG built-in trigger 227
 - SNMP_UNAVAILABLE_CONTEXT built-in trigger 227
 - SNMP_UNKNOWN_CONTEXT built-in trigger 227
 - SNMP_UNKNOWN_ENGINEID built-in trigger 227
 - SNMP_UNKNOWN_USERNAME built-in trigger 227
 - SNMP_UNSUPPORTED_SEC_LEVEL built-in trigger 227
 - SNMP_WRONG_DIGEST built-in trigger 227
 - Source Address (ICMP field) 221
 - standalone operation 23
 - start_at (poll condition function) 174
 - State Definition dialog 247
 - state diagrams
 - icon sizes 252
 - IF-IfColdWarmStart 331
 - IF-IfFramePVC alarm 330
 - IF-IfNmDemand 332
 - IfLoad 51
 - interface-type alarms 329
 - IPSweep 87
 - monitoring loads 29
 - states, defining 245, 247
 - TcpRetransMon 50
 - State/Transition Size dialog 248, 252-253
 - states
 - defining 245, 247
 - deleting 249
 - icons, changing sizes 248
 - status, SNMP v3
 - operations 124
 - StopLookup 96
 - string matching functions 181
 - subnet filter 63, 66, 68
 - subobject scope alarms 43
 - subobjects 205
 - subroutines See Perl 269
 - suppressing nodes 340
 - suppressing polling 340-341
- T -
- TcpRetransMon alarm 50
 - technical support 4
 - test poll 111
 - TestVersion poll 111
 - tips for using property groups and properties 159
 - tools
 - Action Router tool 15
 - tracking conditions 28
 - transition 249
 - defining in an alarm 249
 - Transition Definition dialog 88, 149, 153, 155, 250-251
 - transition line color 253
 - transitions 17
 - actions 240
 - actions, associating 251
 - causing 14
 - defining 249-250
 - deleting 254
 - icon sizes, changing 252
 - Trap Manager error messages 419
 - trap mask trigger function 198
 - defining 198
 - entering notes 210
 - examples 208

- Send Trap alarm action 305, 308
 - trigger function 198
 - variables used 207
 - trap mask trigger functions
 - In() 179
 - trap masks 27, 39
 - attributes 40
 - creating 230
 - data set 39
 - deleting 337
 - enabling 216
 - listing 193, 196
 - using 193
 - trap masks, defining 198
 - traps
 - Inform variable bindings 234
 - trigger function 198
 - built-in triggers 220, 223-225
 - clearing 275
 - delaying 271, 280
 - firing 271
 - firing from an alarm 280
 - functions used for trap masks 203
 - variables used for trap masks 207
 - Trigger Function page 233
 - trigger functions 203
 - creating 202
 - examples 208-210, 235
 - variables 207
 - variables, NerveCenter 301
 - triggers 18, 28
 - attributes 39
 - built-in 220, 225, 228
 - built-in, list of 225
 - data set 39
 - sources 240
 - triggers fired by high-traffic poll 244
 - Type (ICMP field) 221-222
- U -
- understanding NerveCenter 5
 - UNKNOWN_ERROR built-in trigger 227
 - user interface messages 411
 - user name 102, 105
 - using Action Router's object lists 321
- V -
- v3TestPoll 126
 - variable bindings 39
 - attributes 205
 - functions 204
 - Inform alarm action traps 234
 - Inform traps 234
 - NerveCenter Inform traps 234
 - values 205
 - variables 301
 - NerveCenter 301
 - Perl subroutine alarm actions 301
 - trap masks 207
 - trigger functions 207
 - version of SNMP on a node 99

