NORTEL NETWORKS

Preside Multiservice Data Manager

SNMP Surveillance Adapter Guide

241-6001-118

Preside Multiservice Data Manager

SNMP Surveillance Adapter

Guide

Publication: 241-6001-118
Document status: Standard
Document version: 15.1RSUP
Document date: August 2004

Printed in Canada

Publication history

August 2004

15.1 RSUP Standard
Commercial availability except for MPE support which will be available in a future release.

Contents

About this document

The following topics are discussed in this section:

Who should read this document and why

This document is intended for personnel who install, configure, and use the SNMP Surveillance Adapter to monitor Simple Network Management Protocol (SNMP) devices within the network. The SNMP Surveillance Adapter consists of a trap server, data collection daemon (DCD), and SNMP Management Data Router.

> ***Note:*** Data collection daemons can be generic or device-specific. Unless otherwise specified, the DCD referred to throughout this document is the generic DCD.

What you need to know

To use the SNMP Surveillance Adapter, you need to be familiar with the

- Preside Multiservice Data Manager user interface
- UNIX operating system

- Simple Network Management Protocol (SNMP); see “Related documents” (page 17)

How this document is organized

The 241-6001-118 *Preside MDM SNMP Surveillance Adapter Guide* contains many chapters and is intended for different types of users. For information on how to use this document, see the following sections:

- “Documentation roadmap” (page 14)
- “Documentation sections” (page 14)

Documentation roadmap

This document is divided into 10 chapters and three appendices. The first three chapters describe the SNMP Surveillance Adapter and how to install and configure the data collection daemon (DCD) server. These chapters are intended for use by operators responsible for installation.

The fourth chapter describes initial planning required for device integration, including how to model a device and design polling and trap translation. Developers responsible for device integration should begin with this chapter to understand the process. Chapters 5, 6, 7, 8, and 9 provide details required to design polling and trap translation files.

Chapter 10 contains troubleshooting information required for any user. Appendix A contains procedures for generic SNMP device support. Appendix B contains detailed configuration file for a Repeater device. Appendix C contains configuration files required for a dispatcher profile.

The following section provides descriptions and links to the chapters.

Documentation sections

This document contains the following sections:

- “Introduction” (page 19) describes the architecture and functions of the SNMP Surveillance Adapter.
- “Installing and starting the SNMP Surveillance Adapter” (page 31) provides procedures required to set up the SNMP Surveillance Adapter and start the data collection daemon (DCD) server.

- “Configuring the SNMP Surveillance Adapter” (page 37) describes the SNMP Surveillance Adapter configuration files and possible modifications. This section also provides procedures required to configure a new device with SNMP Surveillance Adapter, as well as configuration information for the Network Model and Network Viewer.
- “Understanding the device integration process” (page 65) describes the planning required to integrate a device with the SNMP Surveillance Adapter.
- “Polling and response-handling” (page 83) describes polling request groups, class declarations, and response-handling tasks.
- “Response-handling commands” (page 93) provides descriptions, syntax, and examples for response-handling commands.
- “Command expressions used by the SNMP Surveillance Adapter” (page 125) describes the command expressions that can be used by the SNMP Surveillance Adapter. These include constants, variables, conditional expressions. These also include operators which are described in the next section.
- “Operators used by the SNMP Surveillance Adapter” (page 131) describes the operators that can be used by the SNMP Surveillance Adapter. This section provides operator arguments, functions, errors, and examples.
- “Trap translation configuration” (page 197) describes the trap translation records used in the .tra configuration file. These records consist of trap identification lines, rule labels such as alarm attributes, and expressions.
- “Troubleshooting the SNMP Surveillance Adapter” (page 231) describes different debugging tools and methods. This section also provides log file information, and corrective actions for common error symptoms.
- “Generic SNMP device support” (page 253) provides procedures required to configure a generic SNMP device with the SNMP Surveillance Adapter.
- “Repeater examples” (page 275) shows a device model and configuration file designs for a Repeater device using the SNMP Surveillance Adapter.

- “Dispatcher profile examples” (page 305) shows configuration file designs required for the SNMP Surveillance Adapter to use a dispatcher profile.

What’s new in this document

There have not been any new changes to this NTP for this release.

Text conventions

This document uses the following text conventions:

- `nonproportional spaced plain type`

  Nonproportional spaced plain type represents system generated text or text that appears on your screen.

- **`nonproportional spaced bold type`**

  Nonproportional spaced bold type represents words that you type or that you select on the screen.

- *italics*

  Words that appear in italics in text are for naming.

- `[optional_parameter]`

  Words in square brackets represent optional parameters. The command can be entered with or without the words in the square brackets.

- `<general_term>`

  Words in angle brackets represent variables which are to be replaced with specific values.

- UPPERCASE, lowercase

  Uppercase and lowercase letters that appear in UNIX commands and parameters must be matched exactly. The system matches upper and lowercase characters differently.

- |

  This symbol separates items from which you may select one; for example, ON|OFF indicates that you may specify ON or OFF.

- ...

  Three dots in a command indicate that the parameter may be repeated more than once in succession.

The term absolute pathname refers to the full specification of a path starting from the root directory. Absolute pathnames always begin with the slash (/) symbol. A relative pathname takes the current directory as its starting point, and starts with any alphanumeric character (other than /).

Related documents

See the following documents for related information:

- Course slides: *MDM SNMP Surveillance Toolkit*
- 241-6001-100 *Preside MDM Installation*
- 241-6001-101 *Preside MDM Engineering Guide*
- 241-6001-303 *Preside MDM Administrator Guide*
- 241-6001-310 *Preside MDM Server Reference Guide*
- 241-6001-801 *Preside MDM Overview*
- Internet Standard RFC 1155, *Structure and Identification of Management Information for TCP/IP-based internets*
- Internet Standard RFC 1156, *Management Information Base for Network Management of TCP/IP-based internet*s
- Internet Standard RFC 1157, *A simple Network Management Protocol (SNMP)*
- Internet Standard RFC 1213, *MIB base for Network Management of TCP/IP-based internets (MIBII)*

Chapter 1
Introduction

This section provides an overview of the SNMP Surveillance Adapter and how it works. It contains the following topics:

About the SNMP Surveillance Adapter

The SNMP Surveillance Adapter collects surveillance data from selected devices. It allows surveillance of Simple Network Management Protocol (SNMP) devices to be fully data-driven, and it can run on a Solaris platform without HP Openview.

The SNMP Surveillance Adapter

- is based on the common data collection daemon (DCD) framework defined by the libDcd library
- does not require additional code for support of a new device
- handles responses to polling requests which include discovery, state verification, reachability, and trap-based requests; this response-handling process is specified through device-specific configuration data

- allows alarms to be issued using the configuration language for handling responses to polling requests
- translates traps into alarms; this translation process is specified through device-specific configuration data
- can monitor several device types through the same DCD process
- can run on the same workstation as other DCD processes
- can handle devices with multiple addresses
- can process SNMPv1 and SNMPv2 traps

Architecture of the SNMP Surveillance Adapter

The SNMP Surveillance Adapter consists of a data collection daemon, trap server, and SMDR Management Data Router. For a diagram of the SNMP Surveillance Adapter and items involved, see “Overview of the SNMP Surveillance Adapter” (page 22). For descriptions of the items involved, see the following sections:

- “Data collection daemon (DCD)” (page 20)
- “Trap reporter (TRep)” (page 23)
- “Trap server (TSVR)” (page 23)
- “SNMP Management Data Router (SMDR)” (page 23)
- “General Management Data Router (GMDR)” (page 24)
- “Fault toolset” (page 24)

For more information about these servers, see the 241-6001-310 *Preside MDM Server Reference Guide*.

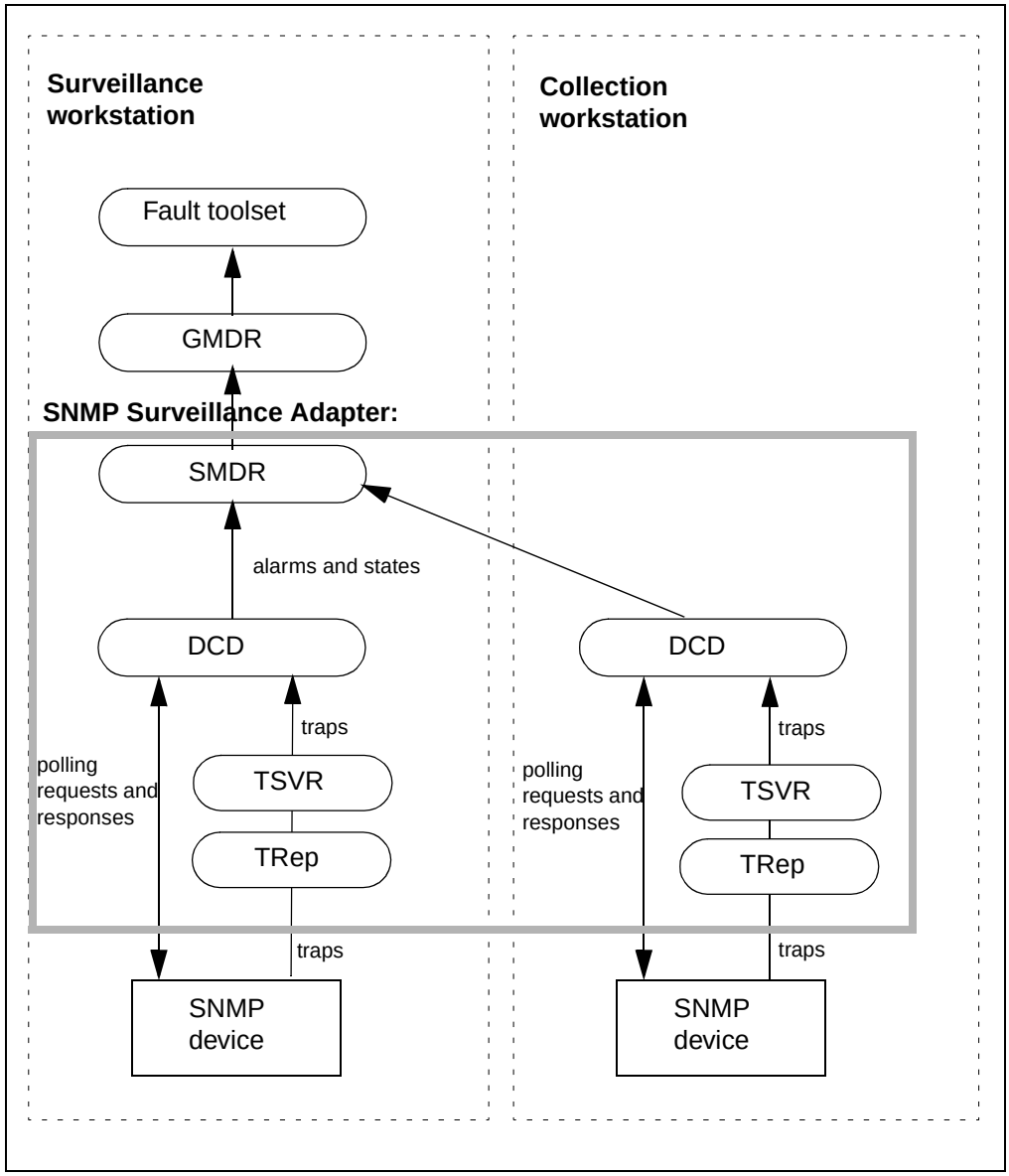
Data collection daemon (DCD)

The SNMP Surveillance Adapter is based on a data collection daemon (DCD). The DCD can be generic or device-specific. Unless otherwise specified, the DCD referred to throughout this document is the generic DCD. It performs the following functions:

- polls the devices using the SNMP protocol to
  - — discover the devices and their subcomponents

  - — verify subcomponent states
  - — verify device reachability
- notifies the SNMP Management Data Router (SMDR) when there is a new component, when a component is deleted, and when there is a change in state of a component
- converts traps received from the trap server (TSVR) to alarms and forwards the alarms to SMDR
- maintains a device seed file containing a list of devices, with their IP addresses and community strings, that is used to rediscover these devices upon restart

Figure 1
Overview of the SNMP Surveillance Adapter



Trap reporter (TRep)

The trap reporter (TRep) receives trap information from SNMP devices and forwards this information to the trap server (TSVR). TRep normally binds to UDP port 162 and cannot co-exist with another process also required to bind to the same port.

TRep's only responsibility is to empty this UDP port input buffer as fast as possible so that no information is discarded. TSVR creates TRep and monitors the state of its connection with TRep. TSVR recreates TRep if it dies. TRep monitors the state of its connection with TSVR and stops if TSVR dies.

Trap server (TSVR)

The trap server (TSVR) receives trap information from the trap reporter (TRep) and forwards this information to the DCD. Before forwarding traps, TSVR filters the traps according to the DCD filter rules. These filters are defined by parameters in the following DCD configuration files:

- .agp for oid filter elements
- .cfg for address filter elements

One trap server services all DCDs on a workstation. For more information about these servers, see the 241-6001-310 *Preside MDM Server Reference Guide*.

SNMP Management Data Router (SMDR)

The SNMP Management Data Router (SMDR) merges SNMP surveillance data obtained from DCDs and makes it available to the General Management Data Router (GMDR).

The SMDR performs the following functions:

- collects surveillance data from generic and device-specific DCDs
- supports REGISTER, GET, CREATE, and ACTION API requests from GMDR
- forwards alarms to GMDR

- issues proxy alarms for the DCD and the device; these alarms are issued when a polling state conflicts with the component active alarm list, or when a device becomes unreachable or reachable again

General Management Data Router (GMDR)

The General Management Data Router (GMDR) collects and stores network surveillance information. This network surveillance information is used by the Preside Multiservice Data Manager (MDM) Fault toolset. The GMDR and SMDR filter duplicate alarm information.

Fault toolset

The MDM Fault toolset consists of a set of tools used for fault management of devices in the network model. It provides text and graphical displays of component state and alarm information. With these tools you can detect, diagnose, and correct faults.

For information about the MDM fault management tools used to display surveillance information, see the 241-6001-011 *Preside MDM Fault Management User Guide*:

DCD output

The data collection daemon (DCD) has the following output:

- “Alarms” (page 24)
- “State change notifications” (page 25)
- “Logs” (page 25)
- “Seed file” (page 25)
- “Server reset notifications” (page 25)

Alarms

The DCD creates alarms in standard MDM format, and indicates the severity of the alarm. See “Alarm attributes” (page 203).

State change notifications

The DCD sends state change notifications every time a component is polled. These notifications contain

- component id
- polled state
- timestamp
- managed state
- customer id. This is a VPN-like numeric code of 0; ignore.

For more information, see “Alarm attributes” (page 203).

Logs

The DCD outputs to a log file or produces standard output (stdOut) depending on the command line option. For more information, see “Log files” (page 247).

Seed file

The DCD appends addressing information to the seed file when new devices are discovered. The DCD recreates the seed file every two hours by default. This time period is configurable; see the 241-6001-310 *Preside MDM Server Reference Guide*.

Server reset notifications

The DCD sends server event notifications that impact more than single components. For more information, see “Server reset notifications” (page 248).

Device surveillance

One of the functions that the Preside Multiservice Data Manager (MDM) performs is surveillance of SNMP devices. As the figure “Overview of the SNMP Surveillance Adapter” (page 22) shows, surveillance information can be transferred directly from an SNMP device or indirectly through a

collection workstation running the data collection daemon (DCD) and trap server (TSVR). The SNMP Surveillance Adapter obtains this information using two methods:

- "Polling" (page 26)
- "Trap reception" (page 27)

Polling

The following steps describe the process of SNMP Surveillance Adapter polling of SNMP devices:

1 The DCD obtains configuration and state information through polling. With polling, the DCD initiates the information flow. It conducts this polling process regularly to maintain the synchronization of configuration and state information between the device and the DCD.

The DCD sends different types of polling requests to the device. Discovery requests are sent once every 12 hours. State verification requests are sent once every 5 minutes. Reachability requests are sent once every 30 seconds. Trap-based requests are sent when triggered by a trap. The frequencies of these requests are configurable. For details, see "Polling request groups" (page 83).

The device SNMP agent gathers the requested configuration and state information and sends a response back to the DCD. If the DCD receives information on a state change, a changed or deleted component, or a new component, it sends a state change notification to SMDR. The SMDR must be registered with the DCD.

2 The SMDR collects the surveillance information that it receives from DCDs. It calculates raw component states and forwards them to the GMDR server.

Using the GMDR Administration Tool, the GMDR server must be registered with the SMDR server in order for GMDR to obtain surveillance information from SMDR. The SMDR can send the same surveillance information into multiple GMDR servers on several workstations by having the GMDR servers registered with SMDR.

3 The GMDR server distributes the alarms and state information to MDM applications such as the Network Viewer Tool and to external applications through GMDR's application interface.

Trap reception

The following steps describe the process of SNMP Surveillance Adapter trap reception for SNMP devices.

1 The DCD receives trap information through the trap server (TSVR). For event notification from traps, the device initiates the information flow.

As the device generates traps, it forwards them to the TSVR. The TSVR forwards the traps to the appropriate DCDs.

2 The DCD translates the traps into MDM alarms and forwards them to the SMDR.

The DCD translates traps into three types of alarms:

SET: These alarms are added to the active alarm list (AAL).

CLEAR: These alarms cause the removal of corresponding active SET alarms.

MSG: These message alarms have no impact on the AAL.

The SMDR then recalculates state information from the new AAL.

3 Based on the contents of the trap, a state poll or a discovery poll can be triggered; see "Polling request groups" (page 83).

4 The SMDR sends surveillance information to the GMDR. The GMDR needs to be registered with the SMDR server in order for GMDR to obtain surveillance information from SMDR. The SMDR can send the same surveillance information into multiple GMDR servers on several workstations by registering the GMDR servers with SMDR.

5 The GMDR server distributes the alarms and status information to MDM applications such as the Alarm Display and Network Viewer Tools, and to external applications through GMDR's application interface.

Device discovery

The data collection daemon (DCD) performs the following types of device discovery:

- "Trap-based discovery" (page 28)
- "Manual discovery" (page 28)
- "Seed file discovery" (page 28)
- "Polling discovery" (page 29)
- "IP discovery" (page 29)

Trap-based discovery

When a trap is received from a new device, device discovery is triggered. The DCD extracts the IP address and community string from the trap and attempts to discover the device. A new device installed in the network will be discovered if it is configured to send traps and accept SNMP requests from the management workstation.

Manual discovery

Device discovery can be manually initiated using the dcdAddNode script. This discovery does not require a restart of the DCD process. This is the script for operators to use when adding a device to the network. For syntax and parameters, see "dcdAddNode" (page 52).

Seed file discovery

The DCD maintains a device seed file that contains a list of devices and their IP addresses and community strings. When the DCD process starts or restarts, the seed file is read and all of the identified devices are automatically rediscovered. The DCD process updates the seed file every two hours. For more information, see the 241-6001-310 *Preside MDM Server Reference Guide*.

> ***Note:*** Do not edit the seed file. Use the manual discovery method to discover new devices.

Polling discovery

If the MIB of a device that has already been discovered contains the information required to discover other devices, this information can be polled. Discovery of these other devices can be triggered from the response-handling configuration associated with these requests.

IP discovery

IP discovery allows devices that are managed by Simple Network Management Protocol (SNMP) to be discovered. A request is sent to the SNMP Surveillance Adapter by querying devices directly to determine their type. IP discovery allows the discovery of neighbouring devices scanning the routing tables of devices as they are encountered. For more information, see 241-6001-011 *Preside MDM Fault Management User Guide*.

Chapter 2
Installing and starting the SNMP Surveillance Adapter

This section explains how to install and start the SNMP Surveillance Adapter. It contains the following topics:

System requirements

There are a number of factors that affect system requirements for surveillance of SNMP devices. For details, see 241-6001-101 *Preside MDM Engineering Guide*

DCD names

The SNMP Surveillance Adapter is based on a generic data collection daemon (DCD). There are three different types of names for any DCD, as follows:

Note: The DCD executable name is also the default process name. This process name can be modified. This customizing is strongly recommended because it allows you to distinguish between DCDs.

DCD server name

The DCD server name refers to the name of the DCD server entered through the Server Administration tool. Configuring the DCD with the this tool requires you to enter the server name in the Descriptive name field of the New Server dialog.

DCD executable name

The DCD executable name is the name of the file found in /opt/MagellanNMS/bin which contains the executable code. The executable name is specified in the startup command; it is `gendcd` for the generic DCD.

DCD process name

The DCD process name is used in the process global options file (.cfg), seed file (.sed) and log file (.log). The DCD executable name is the default DCD process name, but this process name can be modified. When several instances of the DCD run on a workstation, each must be given a distinctive process name. This can be accomplished using the -n and/or -N command line options where:

`-n <name extension>` specifies a name extension to append to the data collection daemon (DCD) executable name to form the process name. This parameter can be used, for example, to distinguish between multiple DCDs monitoring the same device types for different network regions.

For example, if the command is the following:

```
/opt/MagellanNMS/bin/gendcd -n reg1
```

Then the <process name> is: gendcd_reg1

`-N <name prefix>` specifies a name modifier to replace the `gen` substring at the beginning of the executable name to form the process name. This parameter can be used, for example, to distinguish between multiple DCDs monitoring different device types. It allows you to create a device-specific name to replace the `gendcd` string.

For example, if the command is the following:

```
/opt/MagellanNMS/bin/gendcd -N devx
```

Then the <process name> is: devxdcd

For more information, see “Starting the DCD server” (page 35).

Setting up SNMP Surveillance Adapter

CAUTION
Risk of difficulty obtaining fault information
Do not install the SNMP Surveillance Adapter on the same workstation as the HP OpenView desktop feature. Installing HP OpenView and the DCD that is part of the SNMP Surveillance Adapter is not supported.

1. Install the Preside Multiservice Data Manager software on the workstation. See the 241-6001-100 Preside MDM Installation.
2. Start the SNMP Management Data Router (SMDR) if it is not already running. Use the Server Administration tool to start it. For details, see the following:
   - 241-6001-310 *Preside MDM Server Reference Guide*
   - 241-6001-303 *Preside MDM Administrator Guide*
3. Use the GMDR Administration tool to add the SMDR to the GMDR. Depending on your network architecture plan, you can add the SMDR to one or more GMDRs. The GMDR can be local, remote or subordinate. For details, see the following:
   - 241-6001-102 *Preside MDM Planning Guide*
   - 241-6001-303 *Preside MDM Administrator Guide*
4. Use the Server Administration tool to start the trap server (TSVR) if it is not already running. For details, see the following items:
   - 241-6001-310 *Preside MDM Server Reference Guide*
   - 241-6001-303 *Preside MDM Administrator Guide*
5. Use the Server Administration tool to start the data collection daemon (DCD) server. See “Starting the DCD server” (page 35).

6 Select one of the following procedures to identify the DCD to which SMDR is to connect; the second procedure is recommended:

- Add an entry to the server list configuration file /opt/MagellanNMS/cfg/smdr.svr. For details, see 241-6001-310 *Preside MDM Server Reference Guide*
- Run the script smdrCreateServer. This is the recommended procedure. For details, see 241-6001-310 *Preside MDM Server Reference Guide*. The following command example enables the interaction of SMDR and DCD on the local host

  /opt/MagellanNMS/bin/smdrCreateServer gendcd_ott localhost

Note: The `gendcd_ott` parameter in this example represents the process name of this DCD. This command requires the DCD process name, not the DCD server name. For details, see “DCD names” (page 31)

Using the DCD in verification mode

You can run the data collection daemon (DCD) in file verification mode. This feature can be used for debugging purposes. In this mode, the DCD reads the files, issues associated error logs if there are any, and stops with a final log, thus acting as a configuration file compiler

To run the DCD in verification mode, type the following command from a UNIX window:

/opt/MagellanNMS/bin/gendcd [-N <name prefix>] [-n <name extension>] -v [-i]

where:

`-N <name prefix>` specifies a name modifier to replace the `gen` substring at the beginning of the executable name. This forms the process name. This modification also applies to the log file, seed file, and configuration file. Use this parameter to select the configuration files to be verified by the DCD process.

`-n <name extension>` specifies a name extension to append to the DCD executable name to form the process name. This extension also applies to the log file, seed file, and configuration file. Use this parameter when two or more DCD servers of the same type are running on the same workstation

`-v` causes DCD to run in file verification mode.

`-i` causes the system to display logs in the UNIX window instead of sending them to a log file. This optional parameter allows you to see the error logs directly to verify the configuration files.

Starting the DCD server

To start the generic data collection daemon (DCD), type

```
/opt/MagellanNMS/bin/gendcd [-N <name prefix>]
[-n <name extension>]
```

where:

`-N <name prefix>` specifies a name modifier to replace the `gen` substring at the beginning of the executable name to form the process name. This modification also applies to the log file, seed file, and configuration file.

`-n <name extension>` specifies a name extension to append to the DCD executable name to form the process name. This extension also applies to the log file, seed file, and configuration file. Use this parameter when two or more DCD servers of the same type are running on the same workstation.

For more parameter options, see the 241-6001-310 *Preside MDM Server Reference Guide*.

Example 1

To start the DCD with a name extension of reg1, type

```
/opt/MagellanNMS/bin/gendcd -n reg1
```

The process name becomes `gendcd_reg1`. Three default configuration file names change as follows:

- options file: `gendcd_reg1.cfg`
- log file: `gendcd_reg1.log`
- seed file: `gendcd_reg1.sed`

Example 2

To start the DCD with a name prefix of `devx`, type

```
/opt/MagellanNMS/bin/gendcd -N devx
```

The process name becomes `devxdcd`. Three default configuration file names change as follows:

- options file: `devxdcd.cfg`
- log file: `devxdcd.log`
- seed file: `devxdcd.sed`

Chapter 3
Configuring the SNMP Surveillance Adapter

This section explains how to configure the SNMP Surveillance Adapter. It contains the following topics:

Configuration files

The SNMP Surveillance Adapter uses different types of configuration files with preset options. These options can be modified. For configuration files used and their options, see the following sections:

Required file updates

The search priority for each file is as ordered from 1 to 3 in the table "Configuration files and directories" (page 39). If a required configuration file is not in the customer directory location, the directory for other Nortel Networks packages is searched next, and the Preside Multiservice Data Manager directory is searched last.

This means that customer changes take priority. A customer file is used even if a default file with the same name exists.

CAUTION

Risk of losing default file updates

If a customer defines or modifies a file, it is the customer's responsibility to update this file when new loads are installed. The customer file must be updated with any new material present in a newly installed version of the default file supplied by Nortel Networks.

Configuration files and directories

The following table lists configuration files used by the SNMP Surveillance Adapter, and provides the directory locations for each file. For file priority information, see "Required file updates" (page 38).

Note: The definitions for the Preside Multiservice Data Manager (MDM) Network Model schema are stored in the Types configuration files. For locations of these files, see "Network model Types files" (page 58).

Table 1
Configuration files and directories

Configuration file	Origin	Search path
Process global options	1) customer	/opt/MagellanNMS/cfg/dcd/<process name>.cfg
	2) other Nortel Networks packages	/opt/MagellanNMS/ext/lib/cfg/dcd/<process name>.cfg
	3) MDM	/opt/MagellanNMS/lib/cfg/dcd/<process name>.cfg
Agent profile	1) customer	/opt/MagellanNMS/cfg/dcd/<profile name>.agp
	2) other Nortel Networks packages	/opt/MagellanNMS/ext/lib/cfg/dcd/<profile name>.agp
	3) MDM	/opt/MagellanNMS/lib/cfg/dcd/<profile name>.agp
Polling configuration	1) customer	/opt/MagellanNMS/cfg/dcd/<profile name>.pol
	2) other Nortel Networks packages	/opt/MagellanNMS/ext/lib/cfg/dcd/<profile name>.pol
	3) MDM	/opt/MagellanNMS/lib/cfg/dcd/<profile name>.pol
Trap translation	1) customer	/opt/MagellanNMS/cfg/dcd/<profile name>.tra
	2) other Nortel Networks packages	/opt/MagellanNMS/ext/lib/cfg/dcd/<profile name>.tra
	3) MDM	/opt/MagellanNMS/lib/cfg/dcd/<profile name>.tra
Seed		/opt/MagellanNMS/cfg/<process name>.sed
Log		/opt/MagellanNMS/data/<process name>.log

Note 1: The process name is the DCD executable name with optional modifications by the -n and -N parameters. The default process name is gendcd.

Note 2: The profile name is the name specified in the agentProf option in the process global options file.

Process global options file (.cfg)

The process global options file (.cfg) specifies run-time options that are applied globally to all device types. One .cfg file can name many agent profiles; this causes the corresponding agent profile (.agp) files to be

subsequently read. For most configuration parameters in the .cfg file, the default values are adequate. You normally specify only the following run-time options in this file:

- address filters, if required
- agent profiles supported by the data collection daemon (DCD) process

For details, see “Run-time options” (page 41).

Agent profile file (.agp)

The agent profile file (.agp) specifies run-time options for a given device type. For most configuration parameters, the default values are adequate. Specify the following run-time options in this file:

- device identification, which includes device type, trap OID filter elements, agent type, and top-level component category. These device identification parameters are mandatory.
- timers, which are the default value overrides for the specified device type. Most of the time the default values are adequate.

For details, see “Run-time options” (page 41).

Polling configuration file (.pol)

The polling configuration file (.pol) specifies polling requests to send for a given device type, and commands to handle the response. For details, see “Polling and response-handling” (page 83).

Trap translation file (.tra)

The trap translation file (.tra) specifies rules to translate traps into alarms for a given device type. There must be one trap translation record for each trap type.

The syntax for translation rules is similar to the syntax for response-handling commands. For details, see “Trap translation configuration” (page 197).

Seed file (.sed)

The seed file (.sed) contains addressing information for known devices. This includes a list of devices and their IP addresses and community strings. When the DCD process starts or restarts, the seed file is read and all of the identified

devices are automatically rediscovered. The DCD process appends new addressing information when a new device is discovered, and it updates the seed file every two hours.

Log file (.log)

The log file (.log) is not a configuration file, but it contains logs identifying which configuration files were read and errors detected while reading the files if any. For more information, see “Log files” (page 247).

Run-time options

You can modify run-time options in the configuration file (.cfg) and/or in the agent profile (.agp) file. The tables in “Summary of run-time options” (page 43) list the configuration parameters and the files in which they can be edited.

Some configuration parameters can also be specified in the data collection daemon (DCD) startup command. If you specify the same configuration parameter in both the startup command and the run-time options file, the value in the run-time options file is used.

There are global default values for the configuration parameters specified in the DCD framework code. The values can be modified in the run-time options .cfg file for all devices, and the .agp file for a specific device. If device-specific definitions are not defined in the .agp file, the generic DCD uses the values in the .cfg file.

There is one line in the run-time options file for each configuration parameter that you override. The format of each line is
`<keyword>:<value>`
where:
`keyword` is the configuration parameter
`value` is the value you assign to the configuration parameter

Configuration parameters that can be edited in the run-time options file include the following:

- “addrFilter” (page 44)
- “agentProf” (page 45)

- “trapDisabled” (page 51)
- “transFile” (page 51)
- “useAgentAddr” (page 52)

Summary of run-time options

Some run-time options can be modified in the configuration file (.cfg) only. Some run-time options can be modified in the agent profile file (.agp) only. Some run-time options can be modified in the both the .cfg file and the .agp file. The following three tables list the configuration parameters that can be modified, and shows the files in which they can be modified.

Table 2

Configuration parameters that can be modified in the .cfg file only	
addrFilter	agentProf
autoObjDel	congLevel
defaultComm	ignoreTrapCom
objDelInt	seedFileInt
snmpRetrCnt	trapDisabled
useAgentAddr	

Table 3

Configuration parameters that can be modified in the .agp file only	
agentType	deviceType
multiPollAddr	multiTrapAddr
nodeType	portOverride
reachAddrTry	snmpVersion
stPollDelay	dynamicType

Table 4

Configuration parameters that can be modified in both the .cfg and .agp files	
cache	cfgPollInt
defaultPort	disableGetBulk
maxStPollInt	minStPollInt
oidFilter	pollFile
reachPollInt	snmpRespInt
transFile	

addrFilter

The addrFilter option is a character string that you use to filter the IP addresses of devices monitored by this DCD. The DCD rejects any trap or API request that does not pass this filter. Address filters are used to split the network devices between several DCD processes. The size of your network can be a reason to update addrFilter. You can have multiple occurrences of addrFilter if you want to cover more than one subnetwork.

The format of addrFilter depends on your address plan.

Classless subnet addressing

If your address plan is based on classless addressing, the format of addrFilter is:

`a.b.c.d/n`

where:

`a.b.c.d` is any valid IP address
`n` is the number of bits in the specified address that must be matched for the filter to succeed

For example, an IP address will match the address filter, 47.128.154.215/12 if its first twelve bits match the first twelve bits of the address 47.128.154.215.

Classful subnet addressing

If your address plan is based on classful addressing, the format of addrFilter is:

```
<IPelement>[.<IPelement>[.<IPelement>[.<IPelement>]]]
```

where:

`IPelement` represents one of the four positions of an IP address and can be in one of the following formats:

- * accepts all values between 0 and 255
- integer accepts only this value, which must be between 0 and 255
- integer-integer accepts only this range, which must be between 0 and 255

The default value of this parameter is to accept all IP addresses.

Example for classful subnet addressing

```
addrFilter: 55.123.10-40.*
```

This IP address filter would accept any address that has all of the following characteristics:

- the first two elements are 55.123
- the third element is a value between 10 and 40
- the fourth element is a value between 0 and 255

agentProf

The agentProf parameter specifies an agent profile name in the configuration (.cfg) file. This name is used to derive the agent profile (.agp) file name. The .agp file contains device-specific configuration parameter definitions. You can have multiple occurrences of agentProf.

agentType

The agentType parameter is a string used in the polling (.pol) configuration file to identify the type of device agent to build. This parameter has no default value and must be defined in the agent profile (.agp) configuration file.

autoObjDel

The autoObjDel parameter is a flag that indicates whether devices are automatically deleted if they fail to respond to polls within the time interval defined by the parameter objDelInt. Its default value is TRUE.

cache

The cache parameter specifies the name and value of an entry stored in the process cache. It is useful to store permanent information required in the response-handling process. The cache parameter can be defined in the .agp file to store device-specific information, or in the .cfg file to store information that applies to all device types. The syntax of this option is not case-sensitive:
`cache:<entry name>:<entry value>`

cfgPollInt

The cfgPollInt parameter represents the time, in seconds, between consecutive discovery polls of a device. A discovery poll is the polling process that discovers which subcomponents are part of the device and their attributes. The default value of this parameter is 43,200 (12 hours).

congLevel

The congLevel parameter specifies the maximum number of congested replies before the client (SMDR) is cut off. This parameter protects against slow, disconnected, or non-communicating clients. Its default value is 5,000.

defaultComm

The defaultComm parameter can be used to override the community string contained in traps, which is useful in a situation where the community string contained in the trap is different than the string required for device discovery. The defaultComm parameter can supply a value other than the initialized "public". This option must be specified in the configuration (.cfg) file, and it applies to all devices monitored by the process. The syntax is
`defaultComm: <default community string>`

> ***Note:*** To use the defaultComm parameter, the ignoreTrapComm parameter must be set to TRUE.

defaultPort

The defaultPort parameter specifies a value used for the device SNMP agent port number. The default value of this parameter is 161.

deviceType

The deviceType parameter specifies the integer value assigned to this device type. This parameter has no default value and must be defined in the agent profile (.agp) configuration file.

Device type values defined by customers must be within the range of 900 to 999. Values equal to or below 899 are reserved for Nortel Networks development.

disableGetBulk

The disableGetBulk parameter can instruct the DCD to use GETNEXT requests instead of GETBULK requests; possible values are TRUE or FALSE. Use this parameter for devices that only partially implement the SNMPv2 protocol and do not support the GETBULK request. It can be specified in the agent profile (.agp) configuration file. To disable the GETBULK request, specify TRUE.

dynamicType

The dynamicType parameter represents dynamic subcomponents that are deleted from the Network Model. The dynamic subcomponents are discovered through traps and alarms.The dynamic subcomponents are added to the Network Model when they are out of service (OOS) or in-service troubled (ISTB). When the subcomponent's state changes to in-service (INSV) or unknown (UNK), it is removed from the Network Model.

When the SMDR server connects to the GenDCD server, GenDCD passes the dynamic information and the alarm to SMDR. Each time SMDR receives an alarm from GenDCD, it checks the dynamic information from the alarm attribute list. When the dynamic subcomponent state changes into ISV or UNK, it is deleted from the component tree in SMDR and an notification is sent to GMDR. When the dynamic subcomponent is deleted, the TRANSIENT delete notification is sent to GMDR. The Network Model can overrule this notification, but the SMDR cannot force the Network Model to do so.

The DynamicType runtime option must be specified in the configuration (.agp) file. This option does not support wildcards, and it is not case sensitive. An example of a dynamicType runtime option is `dynamicType: ISP;SECURITY.`

ignoreTrapComm

The ignoreTrapComm parameter can be used to override the community string contained in traps, which is useful in a situation where the community string contained in the trap is different than the string required for device polling. The ignoreTrapComm parameter instructs the DCD to use the process default community string. This string is initialized to “public” but it can be modified by the defaultComm parameter. This option must be specified in the configuration (.cfg) file, and it applies to all devices monitored by the process. To activate this option, specify TRUE.

maxStPollInt

The maxStPollInt parameter represents the normal time interval, in seconds, between consecutive state polls of a device. A state poll is the polling process that verifies the state of subcomponents that were already discovered.The default value of this parameter is 300 (5 minutes).

minStPollInt

The minStPollInt parameter represents the minimum time interval, in seconds, between consecutive state polls of a device. Although several events, such as traps, can cause state polls to occur more frequently than the interval specified in the parameter maxStPollInt, state polls are not allowed to occur more frequently than the value of this parameter. Its default value is 30.

multiPollAddr

The multiPollAddr parameter allows the existence of multiple polling addresses for a device. Polling addresses are contained in incoming traps, and they are used to send polling requests. To allow multiple polling addresses, specify TRUE with this parameter in the .agp device file. The default value is FALSE.

multiTrapAddr

The multiTrapAddr parameter allows the existence of multiple trap addresses for a device. Trap addresses can be contained in incoming traps, but they are not used for polling. To allow multiple trap addresses, specify TRUE with this parameter in the .agp device file. The default value is FALSE.

nodeType

The nodeType parameter defines the category name for the device. This node type is used as the first token of each component identifier for this device type. For example, `SSG` is the category for Shasta devices.

objDelInt

The objDelInt parameter represents a time interval, in seconds. If the parameter autoObjDel is set to TRUE and a device does not respond to polling in the time interval defined by objDelInt, the device is deleted. The default value of this parameter is 259,200 (3 days).

oidFilter

The oidFilter parameter specifies a character string passed to the trap server to filter traps forwarded to this process. There can be many OIDs (object identifiers); the first one should be the enterprise OID. All possible traps must match an oidFilter. This parameter has no default value and must be specified in the agent profile (.agp) file.

The value of an oidFilter attribute is an OID expression used to match the beginning of one or several trap OIDs. Each matching trap OID is said to pass this oidFilter.

For more information, see “Trap identification” (page 201).

pollFile

The pollFile parameter specifies the name of the configuration file used to drive SNMP polling and response handling for these devices. Use this parameter in the following situations only:

- sharing poll files between several profiles
- debugging

For recommended polling file paths, see the table "Configuration files and directories" (page 39).

portOverride

The portOverride parameter can be used to override agent port numbers; an agent port is a device port to which SNMP requests are sent. When the portOverride parameter is set to TRUE, the agent port numbers contained in incoming traps are systematically ignored and the value of defaultPort is used instead. The default of this configuration parameter is TRUE.

reachAddrTry

The reachAddrTry parameter specifies the maximum number of polling addresses to be tried before the device is considered unreachable. The default value is 2, which means that if the current polling address fails, only one other polling address will be tried.

> ***Note:*** If the actual number of available polling addresses is less than the maximum number of addresses specified with this parameter, then only the actual number of polling addresses will be tried.

reachPollInt

The reachPollInt parameter represents the time, in seconds, between consecutive reachability polls. A reachability poll is the polling process that verifies whether the device is still reachable. The default value of this parameter is 30.

seedFileInt

The seedFileInt parameter represents the time, in seconds, between consecutive seed file refreshes. Its default value is 7,200 (2 hours).

snmpRespInt

The snmpRespInt parameter represents the waiting time, in seconds, for a response to an SNMP request. Its default value is 10.

The value of this parameter must be high enough to prevent SNMP response timeout in normal operating conditions. However, the value must not be excessively high because it also determines the length of time to detect device unreachability with the following formula:

time = reachAddrTry x (snmpRetrCnt + 1) x snmpRespInt

snmpRetrCnt

The snmpRetrCnt parameter specifies the maximum number of retries for an SNMP request. Its default value is 2.

snmpVersion

The snmpVersion parameter specifies the SNMP version required to communicate with a specific device; options are v1 and v2. This parameter can be specified in the agent profile (.agp) configuration file. The specified SNMP version is overridden if messages with a different version are received from the device. Its default value is v1.

stPollDelay

The stPollDelay parameter specifies the waiting time, in seconds, before polling a subcomponent after a trap has been received against it. This allows transient problems to be cleared before polling requests are sent. The default value is 5.

trapDisabled

The trapDisabled is an option that is set by default to disable trap-based discovery when the DCD process receives a trap from a device. Although trap-based discovery is often a very useful feature, it can be difficult to support. It can also produce inefficient behaviors in some situations, for example, when a common trap OID is shared by many device types. The trapDisabled option allows the DCD to ignore traps from devices it has not yet discovered. To enable trap discovery, specify FALSE with this parameter. To disable trap discovery, specify TRUE with this parameter.

transFile

The transFile parameter specifies the name of the configuration file containing the information used to translate traps into alarms. Use this parameter in the following situations only:

- sharing translation files between several profiles
- debugging

For recommended trap file paths, see the table “Configuration files and directories” (page 39).

useAgentAddr

The useAgentAddr option indicates whether or not the DCD uses the IP address contained in the trap as the address of the device. By default, the IP address associated with the trap is the address associated with the sender process. However, the trap agent address can be set to a value other than the sender's address. If you specify TRUE with this parameter, the DCD will use the IP address contained in the trap instead of the sender's address as the address of the device.

Note: This parameter only applies to SNMPv1 traps.

DCD action scripts

Action scripts can be used to issue action requests to the data collection daemon (DCD). The following scripts are available:

- "dcdAddNode" (page 52)
- "dcdQueryNodeName" (page 53)
- "dcdTriggerPoll" (page 54)
- "dcdAddAddress" (page 55)
- "dcdDeleteAddress" (page 56)

dcdAddNode

The dcdAddNode script issues a request to an appropriate DCD to discover a device. The syntax of this script is

```
/opt/MagellanNMS/bin/dcdAddNode <nodeType> <devName>
<address> <community> <port> <devType>
```

where:

`nodeType` is the category name for the device. For example, `MPA` is the category for a Passport 4460 device.

`devName` is the name of the device. For example, `NEWDEVICE1` can be the device name for any device.

Note: In the dcdAddNode script, the nodeType and devName parameters do not need to be correct. The DCD can discover the device based on the other parameters in the script; it then discovers the real device name by polling this device.

`address` is the IP address of the device.

`community` is the community string of the device.

`port` is the port number of the device SNMP agent.

`devType` is the value of the device type. This value must match the value in the corresponding agent profile configuration file. Device type values defined by customers must be within the range of 900 to 999. Values equal to or below 899 are reserved for Nortel Networks development. For example, the devType value for Passport 4460 devices is 6; the value for an iBWA 5100 device is 20 to 29 inclusive.

Example

The following example represents the dcdAddNode script for a Passport 4460:

```
dcdAddNode MPA NEWDEVICE1 48.222.5.111 public 161 6
```

Note: For information on the addNode API request, see “Add node” (page 241).

dcdQueryNodeName

The dcdQueryNodeName script issues a request to a DCD to retrieve a device name based on the supplied IP address. The DCDs can identify all the nodes matching the supplied IP address and community string pair. The syntax of this script is

```
/opt/MagellanNMS/bin/dcdQueryNodeName <address>
<community> <devType>
```

where:

`address` is the IP address of the device.

`community` is the community string of the device.

`devType` is the value of the device type.

Example

The following example represents the dcdQueryNodeName script for an iBWA 5100 device:

```
dcdQueryNodeName 48.222.5.111 public 20
```

Note: For information on the queryNodeName API request, see "Query node name" (page 240).

dcdTriggerPoll

The dcdTriggerPoll script issues a request to the DCD to rediscover a device. The four optional parameters included in this script are used to enable on demand polling. You cannot enter one of the optional parameters without entering the parameter or parameters that precede it. The syntax of this script is

```
/opt/MagellanNMS/bin/dcdTriggerPoll <nodeType>
<devName> [<delay time> [<group number> [<component
type> [<distance number>]]]]
```

where:

`nodeType` is the category name for the device.

`devName` is the device name.

`delay time` is an integer expression that specifies the delay, in seconds, before the request is scheduled. If the value of this parameter is 0, the request is scheduled with the current time and is added to the ready request queue. If the value of this parameter is greater than 0, the current time is increased by the specified value and the request is added to the ready request queue. If this parameter is not specified, none of the following parameters can be specified.

`group number` is the identifier of the polling request group that must be scheduled on demand. The only group number that cannot be specified is the trap polling group.If this parameter is not specified, neither of the following parameters can be specified.

`component type` is an integer expression that identifies the request that must be scheduled on demand. When you specify this parameter, only the request group with the matching attribute is scheduled. If this parameter is not specified, the following parameter cannot be specified.

`instance number` is a string expression that specifies the instance to be added to each polled variable table OID to poll a single row. This value must be entered in OID format unless the table has only one integer in the index. If this parameter is not specified, the entire table is polled.

Example

```
dcdTriggerPoll SSG SHASTA10 10 4 1 4.2
```

In the example, the Shasta device will be polled using a request defined for group 4, component type 1 and only the row indexed by 4.2 will be polled. This request is scheduled with a delay of 10 seconds.

Note: For information on the poll API request, see “triggerPoll” (page 241).

dcdAddAddress

The dcdAddAddress script issues a request to add an address to an existing device SNMP agent through the API ACTION request addAddr. This request is routed by the Preside Multiservice Data Manager (MDM) servers to the appropriate DCD process.

Note: This script can only be used on a device that has the multiple polling or multiple trap address option enabled. For details, see multiPollAddr and multiTrapAddr in “Run-time options” (page 41).

The syntax of this script is

```
/opt/MagellanNMS/bin/dcdAddAddress <nodeType>
<devName> <address> <community> <address type>
```

where:

`nodeType` is the category name of the device. For example, `SSG` is the category for a Shasta device.

`devName` is the name of the device. For example, `NNEO` could be the device name for an iBWA 5100 device.

`address` is the IP address to add.

`community` is the community string in the address.

`address type` is the type of address, specified as either poll or trap. A polling address is an address to which polling requests can be sent, and from which traps can be received. A trap address is an address from which traps can be received. Polling requests cannot be sent to a trap address.

Example

```
dcdAddAddress CIS REGION1 48.222.5.111 public poll
```

Note: For information on the addAddress API request, see “Add address” (page 245).

dcdDeleteAddress

The dcdDelete Address script issues a request to remove an address from an existing device SNMP agent through the API ACTION request deleteAddr. This request is routed by the MDM servers to the appropriate DCD process. The syntax of this script is

```
/opt/MagellanNMS/bin/dcdDeleteAddress <nodeType>
<devName> <address> <community> <address type>
```

where:

`nodeType` is the category name of the device. For example, `SSG` is the category for a Shasta device.

`devName` is the name of the device. For example, `NEWIBWA1` can be the device name for an iBWA 5100 device.

`address` is the IP address to remove.

`community` is the community string in the address.

`address type` is the type of address, specified as either poll or trap. A polling address is an address to which polling requests can be sent, and from which traps can be received. A trap address is an address from which traps can be received. Polling requests cannot be sent to a trap address.

Note: You can remove all of the addresses for this device at once by using a wildcard (*) in place of the parameters for IP address, community string, and address type. If you use this option, all addresses for this device will be removed except the current active polling address.

Example

```
dcdDeleteAddress CIS REGION1 48.222.5.111 public poll
```

Note: For information on the deleteAddress API request, see "Delete address" (page 246).

Configuring a new device type with SNMP Surveillance Adapter

Use this procedure to configure a new device type with the SNMP Surveillance Adapter so that traps are monitored and viewed through Preside Multiservice Data Manager (MDM) fault management tools.

1. Write the required .agp, .pol, and .tra files.
2. If necessary, edit the data collection daemon (DCD) run-time options file in /opt/MagellanNMS/cfg/dcd/<process name>.cfg. For details, see "Run-time options" (page 41).
3. Perform the following tasks for each DCD process that will manage the devices:
   a. In the process configuration file (.cfg), add the line

      `agentProf: <profile name>`

      This enables retrieval of the device's .agp file.

   b. Restart the process.
4. To verify the validity of the configuration files used by a given DCD process, perform the following tasks from a Unix window:
   a. Execute the command:

```
/opt/MagellanNMS/bin/gendcd [-n <name ext>] [-N
<name prefix>] -i -v
```

For details, see “Using the DCD in verification mode” (page 34).

b. Fix any errors identified.

5 To configure the SMDR server and add an entry for the DCD, open the server list configuration file:

```
:<host name>:<process name>:::
```

Configuring the Network Model and Network Viewer

The Preside Multiservice Data Manager (MDM) model schema defines the component types that can be modeled, the informational attributes of these components, how these components can be inter-connected through links, and how these components can be organized into an organization structure.

This section contains the following topics:

Network model Types files

The definitions in the MDM Network Model schema are stored in a number of ASCII configuration files called Network Model Types files. These files are specific to, and delivered with, each new MDM release. The file locations are different for MDM originals, second-party customized files, and personal customized files.

MDM original Types files

The following list describes each Types file delivered with MDM:

- /opt/MagellanNMS/lib/model/types/types.atdf

  This attribute types definition file defines what types of attributes can be assigned to the module, subcomponents, and links.

- /opt/MagellanNMS/lib/model/types/types.mtdf

  This module and subcomponent types definition file defines the types of components that can be created in the model, how they are contained in one another, and what attributes they support.

- /opt/MagellanNMS/lib/model/types/types.ltdf

  This link types definition file defines the types of links that can be created in the model, what are their legal endpoints, and what attributes they support.

- /opt/MagellanNMS/lib/model/types/types.otdf

  This organization types definition file defines what types of organization structures can be created in the model, and what types of modules and links they are allowed to organize.

Other Nortel Networks packages' Types files

You can customize the Network Model schema by using second-party customized files or by creating your own customized files. The second-party customized files are created by other Nortel Networks groups and are available in the /opt/MagellanNMS/ext/lib/model/types/ directory.

Customer Types files

If you create your own version of the schema files, use the same extensions as described in the preceding list (.atdf, .mtdf, .ltdf, and .otdf).and store the files in the /opt/MagellanNMS/data/model/types/ directory. Your customized .types files are incremental and need only contain the description of entities added or modified. The resulting model schema is the union of all three sets of schema files. Your customized .types file are loaded last so they override the specification of the other areas.

Customizing the Network Model schema

You can customize the MDM Network Model schema. The allowed changes are as follows:

- adding new attribute types
- adding new module, subcomponent, and link types
- adding new organization structure types

- changing the criticality, explanation and label of existing module types
- adding new legal attributes and legal names to existing module types
- changing the criticality, explanation, sub-type, and label of existing subcomponent types
- adding new legal attributes and legal names to existing subcomponent types
- changing the criticality, explanation, and label of existing link types
- adding new legal attributes and legal endpoints to existing link types

If you customize the Network Model schema, the changes are stored in separate types files from those provided by MDM. These files are as follows:

- /opt/MagellanNMS/data/model/types/<file name>.atdf

  This is the customized attribute types definition file.

- /opt/MagellanNMS/data/model/types/<file name>.mtdf

  This is the customized module and subcomponent types definition file.

- /opt/MagellanNMS/data/model/types/<file name>.ltdf

  This is the customized link types definition file.

- /opt/MagellanNMS/data/model/types/<file name>.otdf

  This is the customized organization types definition file.

There can be multiple files of each type. All of these files are loaded as part of the schema. Some schema changes impact multiple files. The following paragraphs describe these changes.

Adding new attribute types in the types.atdf file implies adding their names to the appropriate module, subcomponent, and link legal attributes lists in types.mtdf and types.ltdf.

In previous versions of the Network Model Schema, you could restrict the model by specifying allowed name patterns for modules and subcomponents, legal endpoint types for links, and the allowed module and link types for an

organizational structure type. Although it is still possible to define legal names and link endpoint types, it may not be the case in the future. The current MDM delivered schema does not use these definitions. You can now give modules and subcomponents almost any name. Links of any type can now be created between any ENDPOINT subcomponents. Finally, all organization structure types can contain instances of all module and link types.

After you make any changes, you need to reload the schema changes. To do so, reload the current model or, if appropriate, re-create the model in the usual manner. Models loaded from an image file (Fast Load format) do not use the modified types because the types they were built with are contained in the image. To access the new types, you need to reload the model from its ASCII/portable format. If the type definitions cannot be read, the model does not load and an error message is sent to the System Log display. It is recommended that you use the *makecurrent* macro (/opt/MagellanNMS/cfg/macros/nms/makecurrent <model name>) when loading a model after changing the schema. More diagnostic information may be displayed.

Alternatively, you can load the schema changes without reloading the model. This method has the advantage of being non-disruptive to surveillance. To load the schema without reloading the model, use the following command:

```
/opt/MagellanNMS/bin/poprest -u
```

where:

`-u` specifies the update option.

If the schema changes cannot be loaded using this method, error messages display and the current model is not validated

Individual Types files

The formats of the individual Types files are basically the same as the master types files. It is therefore useful to look at them for examples of how to define a Network Model Schema type. For syntax and examples of these files, see the 241-6001-015 *Preside MDM Network Model Administrator Guide*.

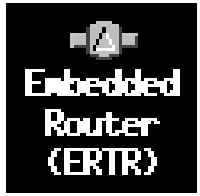
Customizing the Network Viewer

The Network Viewer displays modules in one of two different ways: a type (or flag) icon, and a Shelf icon used in the Shelf Dialog. You cannot create a Shelf Dialog icon for customized module types because this icon needs extensive code support. Modules of a custom-created type are usually displayed with the Unknown Type flag icon.

This section describes how to configure the Network Viewer to display these custom modules with their own distinct type icons.

> ***Note:*** The file /opt/MagellanNMS/data/nvs/icons is available for you to store your own icons.

Node icons in the Network Viewer are defined as XPM3 pixmaps. Each pixmap is identified by a resource line. Another resource line identifies the module type to which the icon corresponds. The following example demonstrates this through the Embedded Router type icon (ERTR node type).



This icon, whose shape resembles a sphere with a pipe going through it, has a triangle inset mark to indicate its relationship with an Access Module (whose general shape is also a triangle). The icon has a 3-D appearance due to the top and bottom drop shadows, and it is colored according to the current node state. The shape of the icon can also be quite complex as XPM pixmaps support the so-called transparent color.

Creating custom node icon pixmaps

An ideal tool for working with XPM pixmaps is the CDE Icon Editor tool (dticon). This tool is suited to the task of creating node icons because of its ability to assign symbolic colors, called Dynamic Colors in the Icon Editor color palette, which is crucial to node icons.

For details about how to use the Icon Editor to create and modify node icons, see the 241-6001-015 *Preside MDM Network Model Administrator Guide*.

Chapter 4
Understanding the device integration process

This section explains the device integration process, and describes what you need to know to plan the integration before writing the configuration files. It contains the following topics:

How to model a device

The device model must be done correctly because all the subsequent steps are based on the model. The following sections describe the device modeling process:

Note: If the SNMP Integrator was being used for a device, a device model was already created for the device. This device model can be used as a basis for the device model required by the SNMP Surveillance Adapter.

Purpose of a device model

The objective of the device modeling activity is to understand the device by collecting information from the device's technical documentation and supported MIBs. Device modeling activity tries to answer the following questions:

- What is the device physical and logical structure?
- Which services does this device provide?
- What are the device components? How are they related to each other? What will be the form taken by their component identifiers?

To create an effective device model, you must determine the principle user groups of the device surveillance information. These principle user groups include the following:

- operators monitoring network behavior
- field technicians repairing faulty equipment
- network OSS archiving information output and producing reports on network behavior

The device model created should resemble these users' perception of the device structure. This enables the effective relating of information produced with the corresponding equipment component.

Modeling a device is logically the first activity required. It does not make sense to do anything else, such as writing configuration files, before gaining a sufficient understanding of the device. The resulting device model, however, can change later if one of the following occurs:

- The MIB does not support the model selected. The establishment of the modeled relationships between components is not possible if the data is not in the MIB.
- New components or services are introduced.

First steps

Before you start to define a device model, select or obtain the values of the following device-level attributes:

- node type
- device type
- agent type

Node type

The node type (or device category) is the component category associated at the top level of the device model with the device itself. The node type is a short character string, preferably two to five characters, occurring at the start of each component identifier associated with one of the device components. The node type selects the icon representing these devices in the Network Viewer, and it is also the root of the network model configuration for these devices. Although the same node type can be used for devices that are closely related, a separate node type should be used for each class of devices requiring a significantly different device model.

If you are modeling a new version of a device that is already supported, use the node type associated with the version that is already supported. Otherwise, select a node type that is not already being used for another device.

Device type

The device type is a number associated with a device type and version. This number is mainly used to select the DCD profile, which is a set of DCD configuration files, used to support a device. Different device models require different device type numbers; different versions of a device may also require different device type numbers if the polling configurations are not compatible.

For devices supported by Nortel Networks, the allocation of device type values is controlled by the Preside Multiservice Data Manager SNMP Base Team. This team has allocated some value ranges to other teams developing configurations for many devices. The range reserved for configurations that are fully developed by customers is from 900 to 999.

If you are modeling a new version of a device that is already supported, you can use its device type temporarily. However, if you determine that your polling configuration is not compatible with the configuration currently used for this device, you should obtain a new device type. If you are modeling a new device, obtain a new device type.

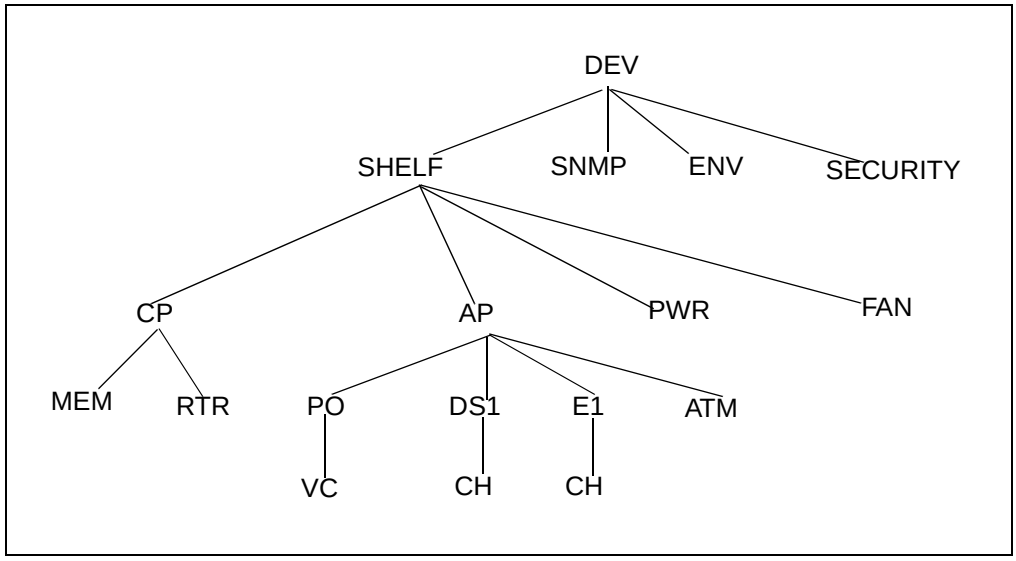
Agent type

The agent type is the name in the polling configuration file associated with the list of polling request types declared for the device. Create a name to identify the device type and version. For example, the profile name could be used.

Initial model

As a first approximation, a device model can be produced based on device physical and/or functional knowledge. This initial model is a starting point; it will be subsequently refined through validation against device MIBs and network operational practices. The following figure shows an example of an initial model for a fictitious device.

Figure 2
Initial model example



where:

- The component category DEV is associated with the device itself. The device includes a maximum of four shelves, an additional board on which the SNMP device agent is running, and another additional board on which an environment monitor is running. Examples of environment factors that can be monitored include intrusion and high temperature. The instance associated with this component is the device name.
- The component category SHELF is associated with shelves. The instance is specified by a number from 1 to 4. The form for the shelf component identifier is: DEV <device name> SHELF <range from 1 to 4>. Each shelf contains one controller processor, up to 8 application processors, up to 2 power supplies and a fan.

  Note: Some level of uniqueness is required for the device category, but reuse of subcomponent categories from one device model to the other is recommended for subcomponents that have the same role conceptually.

- The component category SNMP is associated with the SNMP board. The component instance is $ because there can only be one instance of this component. The component identifier of the SNMP board is DEV XYZ SNMP $.
- The component category ENV and the instance $ are associated with the environment monitor.
- The component category SECURITY (instance $) is not associated with any physical component. It is created to be the target of alarms associated with traps reporting security violations, such as the authenticationFailure generic trap. The SECURITY label is the best choice for these types of traps. Any security violation is important enough to require the network operator to be notified; therefore a MESSAGE alarm is not an optimal choice. However the device is not malfunctioning in any way; therefore a SET alarm against the device itself is not the correct choice.
- The category CP and the instance $ are associated with the control processor. It can have a component identifier such as DEV OTTAWA SHELF 3 CP $. Each control processor has two memory banks and a routing process runs on it. Other processes may also run on the control processor, but this one is identified to show that all components are not necessarily physical.

- The category AP and the instances 1 to 8 are associated with the application processor. These processors can support several types of interfaces: serial port (PO) supporting multiple virtual circuits (VC), channelized DS1, channelized E1, and ATM.

The example shows that relationships between component types are often based on physical or logical containment. These relationships directly define the component identifier syntax. They also define the state propagation rules. These relationships must eventually be encoded in Network Model configuration files.

When physical components are involved, choosing instance values based on the physical location of the components enables the visual identification of a troubled component. The device model then becomes more useful for its intended users.

Model refinement

This section describes how to refine the model by removing components that are not required or cannot be supported.

Some components may not be required because their existence or their state is implicit, and their presence in the device model is not beneficial. For example, the existence and state of the SNMP component can be assumed. The fact that we are able to manage the device proves that it is in-service. If no trap is issued against this component, removing it simplifies the model without losing any information.

Although some components may exist, do not include them in the device model if the MIB does not contain information about them, and no trap is sent to signal a state change. In the example, this may apply to the FAN or the VC components if the device MIB does not contain information about them.

The model structure must be supported by the information available in the MIB. For example, to identify a DS1 channel, you must identify the channel instance for the DS1 channel as well as each parent instance (DS1, AP, SHELF, DEV). You may have to scan several MIB tables during device discovery and cache some information to be retrieved when required. If this is not possible, modify the model to match what the MIB can support. In the example, if you cannot determine the shelf instance associated with a power

supply, model these PWR components under the device itself even though they are associated with individual shelves. Selecting an instance value for these components may then become difficult.

Choosing the polled components

After you create and refine the initial model, you must decide which components will be discovered as part of device discovery, and have their state periodically monitored. These components are referred to as managed or polled components. You must also decide which components will only be discovered when alarms resulting from trap translation will be raised against them. These components are referred to as dynamic components. The decision to only discover a component dynamically may be based on one of the following reasons:

- It is not necessary to monitor this component unless it reports a problem. In the example, this could apply to the ENV, PWR, MEM and RTR components.
- This component is a logical entity used to report a category of problems; the SECURITY component is one of these.
- This component is not very important operationally.
- This component is not likely to fail.
- There are so many instances of this component that monitoring them continuously would degrade the performance of the device or the surveillance system. In the example, this may apply to the CH components.
- These components are created and destroyed frequently, making them unfit for inclusion in a stable device model. In the example, this could apply to the VC components.

After you decide which components can and will be polled, you are ready to design the polling operation for this device type.

How to design polling

The following sections describe the polling design process:

- “Purpose of a polling design” (page 72)
- “Device names” (page 72)

- “Component discovery” (page 73)
- “State verification” (page 74)

Purpose of a polling design

The polling design process identifies which MIB variables or tables are polled to produce the information to discover the polled components and compute their respective state. This process tries to answer the following questions:

- How is the device name defined? Can it be modified, and if so, how?
- What MIB information is required to define the component hierarchy, such as the instances associated with components at all levels? From a given subcomponent, how are the instances associated with its parent components determined?
- Which MIB information is required to compute the state of each component? If several variables are involved for a component, what are the state computation rules?

Polling is a fundamental part of managing an SNMP device. It is the only way to discover which components are present. In most SNMP agent designs, the trap flow does not provide a reliable and sufficient information flow to monitor the health of these components. The purpose of the traps is to trigger additional polling requests to verify the impact of problems reported. Polling is often the only way to find out when a reported problem is resolved.

Device names

The device name is a string that contains the device category, a space, and a device instance identification respectively. This name is used by all the Preside Multiservice Data Manager MDM tools and processes to uniquely identify the device. The device name should enable easy identification of the device for the intended users of the surveillance information. The device instance identification must comply with the following rules:

- maximum length is 35 characters
- contents include only uppercase letters, digits, periods, dashes, and underscores

SNMP devices often use the SysName MIB variable to identify each device. When this is done, the device discovery request must handle the following situations:

- SysName has not been defined yet.
- SysName contents do not produce a valid device name. You can use MAKENAME operator to sanitize the contents.
- SysName contents have been modified since they were last polled.

The SNMP Surveillance Adaptor does not require SysName to be used to name the device. Other possible options are:

- a name obtained from another MIB variable
- a hard-coded name supplied by the user who is likely using the dcdAddNode script
- a network host name discovered using the HOSTNAME operator

Component discovery

SNMP is not designed to produce a device hierarchical representation. A device MIB is mainly composed of one-dimensional tables which sometimes reference other tables by using index values from other tables. When a device is highly structured, rebuilding the structure from such tables is challenging.

Polling must be designed to enable component discovery. This requires identification of the MIB tables in which each of the polled components are enumerated, and identification of which variables are needed to define each component instance and its parent components instance. If the required information is not available in a single table, additional polling requests may be required to previously scan other tables and cache some of their information.

In the previous example, a MIB table may list all the device PO components. Each table entry contains the index of a parent AP component in the corresponding table. The AP table may not be indexed by the slot number, which was chosen as the instance for the AP components, but this information may be available in each entry of the AP table. Then when the PO table is scanned, the instance of the AP parent component is not directly available. Only the index of the table which contains the required value is available. The

problem can be solved by caching the required information when scanning the AP table in the form of cache entries with a “AP<index>” key and a value specifying the component identifier of the AP component (including the slot number instance value). When the AP index is obtained from the PO table entry, the PO component identifier can be formed by retrieving the string cached against the “AP<index>” key and appending to it “PO” and whatever instance value applies to this PO component. This component identifier can then be stored in another cache entry to be subsequently used to define VC component identifiers.

When components are discovered, there may also be a requirement to discover some of their attributes and store those as component properties so that the information can be retrieved by other processes. An example of this might be the serial number of control and application processors that may be required by some inventory application. When designing the polling activity, the known additional polling requirements should be included, however, it is usually simple to handle new requirements of this type when they are discovered.

State verification

Designing state verification consists mainly of

- identifying the MIB variables or tables containing the information required to compute each component state
- formulating rules to produce a state value from any possible combination of these values

The following states are valid:

- INSV: the component is functioning properly, and is inService.
- OOS: the component cannot perform its intended function, and is outOfService.
- ISTB: the component is functional, but has one or more problems that should be addressed, and is InServiceTroubled.

- UNK: the values of the polled variables form a combination that is not considered possible, and the component state is unknown. This state is also assigned to all the components of a device that is currently unreachable, but the component state is not then defined by a polling response.

How to design trap translation

The following sections describe the trap translation design process:

- "Traps and alarms" (page 75)
- "Inventory" (page 77)
- "Translation rules" (page 77)
- "Alarm attributes" (page 77)

Note: If the SNMP Integrator was being used for a device, a device model and trap translations were already designed for the device. These trap translations can be used as a basis for the trap translation design required by the SNMP Surveillance Adapter. Although the basic design can be reused, the translations must be rewritten for the new language. You may also choose to improve or modify the original design.

Traps and alarms

An SNMP trap is an asynchronous message sent by the device to inform the management applications of the occurrence of important device events. In the original SNMP design, the purpose of a trap was to trigger additional polling activities to establish the precise nature and impact of the event. Most traps report the occurrence of problems. Some traps report the resolution of a problem or some other operational information. A trap is composed of the following elements:

- a header which contains
  - addressing information
  - problem identification codes (SNMPv1)
  - security information (SNMPv3)
- a list of variables which are values extracted from the MIB to supply additional information about the event reported

Note: All SNMPv2 traps contain two implicit variables always inserted as the first two in the list: sysUptime and notificationID. Therefore if the SNMPv1 and SNMPv2c of the same trap are compared, the first variable in the SNMPv1 trap (index 0) is the third variable (index 2) in the SNMPv2c trap. The position of each subsequent variable is also shifted by two.

Preside Multiservice Data Manager (MDM) alarms are also device event reports. Alarms are created or forwarded by surveillance processes to provide information about

- the occurrence of problems (SET alarms)
- the resolution of problems (CLEAR alarms)
- other operational information (MSG alarms)

An alarm contains a set of attributes describing the nature of the event reported and the component that is directly affected. A SET alarm is considered active if no CLR alarm has yet been received to signal the resolution of the problem it reports. For each component, MDM maintains an active alarm list from which the component's raw state is calculated. The alarm flow may be the main information source used to monitor a customer's network. This alarm flow must provide an accurate representation of the network's health.

This process of designing trap translation tries to identify the rules and the information required to translate traps into alarms. This process requires a good understanding of the following:

- the impact of different hardware failues on the services provided by the device
- the service-critical aspects of the device from a customer perspective
- the severity of problems identified by traps received

Inventory

You must determine which traps can be sent by the device. These include the following:

- generic traps
- traps defined in a standard MIB supported by the device agent, such as a repeater MIB or a frame relay MIB
- any trap defined in a proprietary MIB supported by the device agent

When this inventory is complete, you must determine the contents of these traps. This is the main source of information available to the trap translation process.

Translation rules

To generate each alarm attribute, use the following information sources:

- trap variables (main information source)
- cached information discovered by polling; however, you must provide for the possibility that this information may not have been discovered yet when the trap is received
- device documentation from published MIBs or from other technical documents
- device development or support group

Several traps can be translated into the same alarm if they describe the same event. Conversely, the same trap can be translated into one of several possible alarms based on the contents of some variables.

Alarm attributes

For each trap type, you must define a rule to produce each of the required alarm attributes. See the following alarm attributes.

Note: For more information and other alarm attributes, see “Alarm attributes” (page 203).

Component identifier

The component identifier attribute is often obtained from cached information based on MIB table indexes stored in trap variables. It is important to raise the alarm against the correct component so that the source of the problem is clearly identified and the alarm has the correct state impact. Occasionally the nature of the trap itself directly identifies the affected component; this is often true when there is a single instance of a component on the device, such as a single fan or power supply.

Timestamp

Traps usually do not contain variables that can be used to generate a timestamp. The corresponding rule is then omitted and the workstation time is used.

Fault code

The fault code value normally comes from the fault code allocation plan designed during this device integration task. This fault code is an eight hexadecimal character code used as an index into the alarm Nortel Networks technical publication (NTP) to access contextual information for each alarm. Usually the first four characters identify a fault code range assigned to a specific device type, and the last four characters identify the problem using a device-specific coding.

For each device type, you must design a fault code allocation plan used to generate values for this attribute. For details, see “Fault_code” (page 205).

Trap translation records can also be introduced by customers to

- supplement those defined by Nortel Networks for a given device. Follow the conventions listed in “Fault_code” (page 205) to avoid duplicating fault codes.
- translate traps received from a device fully supported by the customer. Follow the conventions listed in “Customer-defined alarms” (page 205).

Severity

For CLEAR alarms, the severity should be “cleared”. For MESSAGE alarms, severity can be anything but “cleared”, including “indeterminate”. For SET alarms, severity must be “critical”, “major”, “minor” or “warning”. Alarms with “critical” or “major” severity put components in the OOS state. A “critical” severity indicates that operator intervention is required, while a

"major" severity normally indicates that the problem may be automatically corrected. Alarms with "minor" or "warning" severity put components in the ISTB state unless a more severe alarm is also active against the same component.

Comment, alarm type, probable cause

The trap often does not contain any explicit information to define the comment, alarm type, and probable cause attributes. These attributes must be hard-coded based on information obtained from the device documentation or the device group. The comment attribute should contain a brief and meaningful description of the problem, supplemented by run-time data if it is available from the trap variable. The Nortel Networks technical publication (NTP) normally contains a more detailed description.

Event

Normally, the event attribute value is based on the following rules:

- If the trap signals the occurrence of a problem, it is a SET alarm.
- If the trap signals the resolution of a problem, it is a CLEAR alarm. Then the fault code selected is the same as the fault code assigned to the corresponding SET alarm.
- If the trap only contains additional information useful for the network operator, it should be MESSAGE alarm which does not impact the component state.

Exceptions to these rules are as follows:

- If the problem reported is not important or can be quickly fixed by the device auto-correcting itself, then MESSAGE may be a better choice. The information is still made available, and it does not draw the operator's attention away from problems requiring intervention.
- If the information conveyed by the trap is about some component (or the device itself) being restarted or reinitialized, CLEAR is a better choice. The clearScope attribute should be defined to cause the removal of all the alarms active against this component and its subcomponents before the restart.

Clear scope

Normally, a CLEAR alarm removes a SET alarm active against the same component and having the same fault code. It can be necessary to widen the scope of some CLEAR alarms so that they can be used to remove more than one active SET alarm. The clrScope alarm attribute is used to do this; it only applies to CLEAR alarms. The two main options are as follows:

- clearBase: the CLEAR alarm only applies to the same component, but it clears all the alarms currently active against this component.
- clearHier: the CLEAR alarm applies to the component and all its subcomponents. All alarms active against one of those components are cleared. An obvious example of this option is the alarm generated by a cold start trap. Such an alarm should clear the alarms active against all the device components.

Alarm age

The AlmAgeMin and AlmAge attributes allow old outstanding alarms to be cleared. These attributes cause an alarm to be discarded if it does not reoccur after a specified period of time. The AlmAgeMin setting is in minutes. The AlmAge setting is in hours. Use one of these attributes if a SET alarm is not expected to be cleared by a trap signaling the resolution of the problem.

The minimum value allowed for the AlmAgeMin setting is four minutes and the minimum value allowed for the AlmAge attribute is one hour.

If the same alarm is issued against the same component before it has reached this aging time limit, the waiting period is reset. The alarm is cleared only after it has been active for the specified period from the last repetition.

The SNMP Surveillance Adapter adds the AlmAgeMin or AlmAge attribute to the alarm obtained from the trap translation. When an alarm with the AlmAgeMin or AlmAge attribute is stored in the SNMP Management Data Router (SMDR), SMDR creates a timer for the alarm. When the alarm has aged, the timer triggers SMDR to clear the alarm and SMDR issues the corresponding CLEAR alarm.

Note: Use aging attributes only for alarms issued against dynamic components, which are components that are not polled. Active alarms for polled components are cleared by the first polling operation that discovers the component state is back to normal. This discovery usually occurs long before the alarm has aged.

Integrating a device driver for use with IP Discovery

A data collection daemon (DCD) driver that needs to become visible to the Internet Protocol (IP) Discovery application needs to add one or both of the following flags, embedded in a comment line, to the agent profile (.agpfile):

- #sysObjectID:
- #dispType:

Note: The flags are case sensitive.

The #sysObjectID: flag indicates which sysObjectID(s) the agent will manage. The format for a line specifying the #sysObjectID: flag is as follows:

#sysObjectId: <sysObjectID>[;<sysObjectID...]

There may be more than one sysObjectID line in an agent profile. Each sysObjectID corresponds to one or more sysObjectIDs that the agent is managing. The sysObjectID value cannot have a wildcard (*) in it, but it may have a partial match to the sysObjectID. Therefore, a sysObjectID value of 1.3.6 matches all values beginning with the prefix 1.3.6. If the sysObjectID of a device is unavailable or not meaningful for discovery, but the device type needs to be visible to the IP Discovery application for use with the Discover As functionality, then specify a sysObjectID value of “none”. For example, #sysObjectID:none.

The #disType: flag specifies the device type associated with the agent profile. The value of the flag is displayed on the IP Discovery user interface as the device type for this driver. If the #dispType: flag is missing from the agent profile, but the sysObjectID flag is present, then the value specified by the agentType configuration parameter is used to identify the device type for a driver. If the disType flag is present, but the sysObjectID flag is missing, then a value of “none” is assumed for sysObjectID used during discovery. Spaces are not permitted in the value specified for the dispType: flag.

The #disType: flag is also useful for dispatcher profile processing. If this flag is used in two or more different agent profiles with identical dispType values, then it creates a method to identify the agent profiles with one unique device type. Multiple profiles referenced by a common dispatcher profile may be grouped and managed under a common descriptive name, by specifying a common dispType value.

Only agent profiles which include one of these flags are visible to the IP Discovery process. By adding or omitting these flags, device integration groups can control which device types are visible to the IP Discovery application.

For more information, see the 241-6001-011 *Preside MDM Fault Management User Guide*.

Chapter 5
Polling and response-handling

This section describes the command format and rules for polling and response handling. It contains the following topics:

Note: The examples in this section are specific to a particular release level of the SNMP Surveillance Adapter framework and a specific release level of a device. Actual system output can change due to framework enhancements and device updates.

Polling request groups

The SNMP Surveillance Adapter is based on a generic data collection daemon (DCD) that collects traps from SNMP devices. The DCD also polls the devices regularly to maintain its synchronization of configuration and state information with the devices. If the DCD receives information on a state change, a changed or deleted component, or a new component, it sends a notification to SMDR.

The DCD sends different types of polling requests at different at preset time intervals. These time intervals can be modified. See the following sections for types of polling requests sent and their time intervals.

- "State verification requests" (page 84)
- "Reachability requests" (page 84)
- "Trap-based requests" (page 85)

The also sends out polling requests on demand, instead of at preset time intervals. See "On-demand requests" (page 85) for more information.

Discovery requests

Discovery requests are a set of polling requests used to scan the MIB tables and create discovered components. By default, discovery requests are sent every 12 hours. To reset this value, see cfgPollInt in "Run-time options" (page 41). In addition to the default, the discovery requests are sent when

- configuration changes are detected by polling
- an operator requests device discovery using dcdAddNode, or device rediscovery using dcdTriggerPoll
- trap translation or response handling triggers the rediscovery
- reconnection triggers the rediscovery

State verification requests

State verification requests are a set of polling requests used to scan the MIB tables and refresh the component states. State change notifications and proxy alarms are issued by SMDR if required. By default, component states are refreshed every five minutes. To reset this value, see MaxStPollInt in "Run-time options" (page 41). In addition, component states are also refreshed when

- an operator requests state verification
- trap translation triggers state verification

Reachability requests

A reachability request is a single request used to verify if the device is still reachable. It usually polls a single variable, which is often sysName so that this request can also be used to detect device name changes. The response-handling command block does not need to do anything; the purpose of this request is to receive a response. The presence of a reachability request is mandatory; otherwise device reachability cannot be established or verified.

By default, this request is sent every 30 seconds. To reset this value, see reachPollInt in “Run-time options” (page 41).

Trap-based requests

When the DCD receives a trap that impacts a polled component state, this component is polled immediately to confirm the state change. No request has to be defined to do this; the polling request sent is the corresponding state polling request modified to poll a single component.

On-demand requests

The on-demand polling group is different from the other four polling groups because the requests in the on-demand group are scheduled to be sent out at specific times. They are scheduled by the Poll command

- the Poll command, in response to handling or trap translation configurations
- the dcdTrigger script or corresponding API action request

Response-handling process

When a response is received, the following process occurs through the DCD:

1 If the METHOD attribute is table, the DCD first checks if the end of the table has been reached. If the end is reached, the response is discarded. The DCD deletes the components of this type that are no longer found in the table.

2 The DCD verifies the responses to determine if any requested variables are missing. If variables are missing, an error log is issued and the response is discarded.

3 The response information is extracted and stored in a response-handling context. The first response-handling command block is executed in this context. If there is more than one block, this block must select the correct block through Next or UseBlock commands.

Response-handling tasks

Configuration data in each polling class declaration specifies how responses to polling requests are handled. This response-handling configuration language allows the following tasks to be data driven:

- extracting data from the reply. The configuration data describing the request already specifies which variables the reply should contain. This task consists of retrieving the information from the response and is handled by the framework when creating the context.
- deciding whether or not to discard the response. Sometimes the response should be ignored, such as when a table entry corresponds to a component that is not modelled.
- deciding how to handle the response. The response contents may determine which way the response is handled. For example, there can be an entry from a port table grouping several port types, each of which is handled a specific way.
- assembling the component identifier. This task often requires the most work. The task can involve accessing information previously discovered, extracting and parsing response information, and assembling several parts of the component identifier string.
- computing the component new state. The values of some response variables are used to compute a new state and update the component.
- storing some component properties. This task involves storing the values of some response variables. Client processes can use the queryProperty requests to access these variables.
- caching some information. This task involves storing some of the discovered information to handle subsequent responses.
- triggering device rediscovery. If response information indicates that the device configuration has changed, a new discovery polling cycle can be triggered.

Polling class declarations

In the DCD framework, the polling configuration file (<profile name>.pol) describes the contents of all the polling requests sent to devices of a given type. This polling configuration file contains two types of top-level declarations:

- polling class, introduced by the keyword CLASS. Each of these declarations defines a separate polling request type.
- agent type, introduced by the keyword TYPE. Each of these declarations specifies the list of polling requests sent to the associated device type. The list of polling requests consists of polling class names.

The following sections describe polling class declarations:

- "Structure of a CLASS declaration" (page 87)
- "Example of a CLASS declaration" (page 90)

Structure of a CLASS declaration

A CLASS declaration must be associated with each polling request sent to a device. It has the following structure:

```
CLASS <class name>

{

  REQGROUP <group number>
  COMPTYPE <component type number>
  METHOD   single | table

  <polled variable declarations>

  <command blocks list>
}
```

Class name

The CLASS name is the name associated with the request declared. This name is used in TYPE declarations to refer to the request. The name must be unique within the .pol file containing the CLASS declarations; it can take the form of any character string without spaces. The recommended practice is to use a name describing what the request does, such as *mib2DeviceDiscovery*.

Group number

The group number identifies the polling request group, including the declared request, as follows:

- 0: discovery
- 1: state verification
- 2: reachability check
- 3: trap-based polling
- 4: on-demand polling

The value that identifies trap-based polling never occurs in CLASS declarations, but it is used internally when such a request is derived from the state verification request with the same COMPTYPE value.

Component type number

The component type number is assigned to the components associated with the declared request. Within a request group, requests are sent in the order defined by this number. When a polled component is created by a discovery request, a component property is also created and contains this value; the name of this property is *NortelMomsObjectTypeId*. Discovery and state verification requests associated with the same components should have the same COMPTYPE value.

The component type number assigned to the device (top level) component must be the lowest COMPTYPE value for each agent type. This makes it the first discovery request sent, and allows the DCD to identify the discovery request dealing with this top-level component. The CompId command contained in a device discovery request is handled differently compared to CompId commands in other requests.

Method declaration

This request attribute specifies whether the SNMP request sent is a GET (single) request or a GETNEXT (table) request. For SNMPv2 devices, GETBULK requests are used instead of GETNEXT requests to reduce the load on the device as well as the communication costs. Optionally, the configuration may require GETNEXT requests to also be sent to SNMPv2 devices. See “disableGetBulk” (page 47).

Polled variables

This section of the CLASS declaration lists all the variables polled by the request. A polled variable has the following syntax:

```
<variable name> <variable OID>
```

where:

`variable name` is any identifier subsequently used to represent the value returned in the response in this request command block. You do not need to use the name associated with the same variable in the device MIB, although an abbreviated version is recommended for mnemonic purposes.

`variable OID` is an SNMP object identifier in numeric format. This OID is either the OID of the MIB variable instance required (METHOD single), or the OID of the MIB table column scanned (METHOD table). When a table is polled, all the variables must refer to table columns indexed by the same variable(s).

Command blocks

Command blocks specify how to perform the required tasks when a response to a request described by this CLASS declaration is received. The list of tasks to perform is often short and each task is relatively simple.

Commands forming commands blocks each start on a new line and have the following syntax:

```
<command name>: <command argument list>
```

where:

`command name` identifies the command to be executed. For supported commands, see “Response-handling commands” (page 93).

`command argument list` is a list of expressions that specify the value of each parameter required by the command. For descriptions of parameters required by each command, see “Response-handling commands” (page 93). For syntax of the expressions used, see “Command expressions used by the SNMP Surveillance Adapter” (page 125) and “Operators used by the SNMP Surveillance Adapter” (page 131).

If more than one line is required for a command, each additional line must be introduced by a '\' at the end of the previous line. The line break can occur at any place where a blank character could occur. The first non-blank character of the continuation line is always separated by exactly one blank from the last non-blank character of the previous line, excluding the terminating '\'.

Command blocks are separated by blank lines. Each block specifies one of the alternate ways in which the response can be handled. For most responses, only one block is required.

Example of a CLASS declaration

The following polling class declaration defines a request to update the state property of previously discovered board components. This update is performed through polling of the corresponding table.

```
CLASS edgeBoardState
{
  REQGROUP 1
  COMPTYPE 3
  METHOD table

  # polling variables
  board   1.3.6.1.4.1.562.12.1.1.1.1.1
  status  1.3.6.1.4.1.562.12.1.1.1.1.8

  # response handling command block
  Assign: name CACHE(CAT("BOARD", board))
  Discover: EMPTY(name)
  CompId: name
  State: [ EQ(status, 1) ? INSV \
           EQ(status, 2) ? OOS \
           UNK ]
}
```

This response-handling command block contains the following commands:

1 Search the cache for an entry created previously with the name BOARD concatenated with the value of the board polled variable. This entry must have been created in the last discovery polling cycle. If the entry is found, assign the corresponding value to the local variable name. If the entry is not found, the value assigned to the name is the empty string.

2 If the value of name is the empty string, schedule a discovery polling cycle. This cycle is required because the table contains a previously undiscovered entry. Discard the response and discover the component first.

3 Define the component identifier as the current value of name.

4 Update the component state based on the value of the status polled variable. If the value is 1, the component is in service. If the value is 2, the component is out-of-service. If the value is other than 1 or 2, the new state is unknown.

Directives

The polling configuration file (<profile name>.pol) can contain include directives. Include directives instruct the DCD to include file declarations that are contained in another file. They can be used to ensure that several agent types use common declarations to do certain tasks. Include directives have the following syntax:

```
#include <file name>
```

where:

`file name` identifies the file to be included. If `<file name>` does not include the directory path, a search priority is used. If a directory path is included, the search priority is not used. For more information about the search priority, see “Configuration files and directories” (page 38).

Included files can also contain include directives:

- Files included in .pol files using this directive can only contain complete CLASS or TYPE declarations.
- Include directives must precede all CLASS and TYPE declarations in the file.

When an include directive is used to introduce common CLASS declarations, the corresponding CLASSNAME must be added to the TYPE declaration so that common polling classes are used by the corresponding agent type(s). For example, if the file common.pol defines the common polling classes abcdDisc and abcdState, these classes can be included in the file xyz.pol as follows:

```
#include common.pol

CLASS <declarations specific to xyz.pol>

TYPE xyzAgent
{
CLASSNAME <xyz.pol specific class name>

CLASSNAME abcdDisc
CLASSNAME abcdState
```

Chapter 6
Response-handling commands

This section describes SNMP Surveillance Adapter commands. It contains the following topics:

Note: The command examples in this section are specific to a particular release level of the SNMP Surveillance Adapter framework and a specific release level of a device. Actual system output can change due to framework enhancements and device updates.

Summary of response-handling commands

The following table lists commands that can be used in a response-handling command block.

Table 5
Response-handling commands

Command	Purpose
Next	To conditionally jump to the beginning of the next block
UseBlock	To conditionally execute commands in the current block
Quit	To conditionally stop processing the response
Log	To issue a log
Discover	To conditionally schedule a new discovery polling cycle and stop processing the response
Poll	To conditionally schedule a new state polling cycle
Assign	To assign a string value to a local variable
Compid	To assign a value to the component identifier
State	To update the component state
Cache	To create or update a cache entry
Property	To create or update a component property
Alarm	To issue an alarm triggered by polling results
addAddr	To add an address to a device's SNMP agent
clearAddr	To remove an address from a device's SNMP agent
NewDev	To discover a new device immediately

Next command

The Next command is used to jump to the next block if an associated condition indicates that the current block does not apply to this response instance. The configuration data that describes how to handle the response can be divided into command blocks. Each block specifies an alternate method of handling the response.

> ***Note:*** The Next and the UseBlock commands both allow you to select the correct command block to handle a response. However, the conditions in these two commands have opposite effects. The Next command jumps to the next block if the condition evaluates to true. The UseBlock command continues to execute commands in the same block if the condition evaluates to true.

Syntax

```
Next: <condition>
```

Parameters

`condition` A Boolean expression that indicates the condition for the next jump. If this condition is met, the jump to the next block occurs. If this condition is not met, the process continues with the commands in the same block.

Example

The following polling class declaration discovers device interfaces by polling the MIB-II interfaces table. The class contains several response-handling blocks that are each designed to handle a different interface type. The correct block is selected by a sequence of Next commands that test the value of the ifType response variable.

```
CLASS ifTableConf
{
  REQGROUP 0
  COMPTYPE 3
  METHOD table

  ifIndex    1.3.6.1.2.1.2.2.1.1
  ifType     1.3.6.1.2.1.2.2.1.3
  ...
```

```
  Next: NE(ifType, 30)
  # block for DS3 interfaces
  ......

  Next: NE(ifType, 39)
  # block for SONET interfaces
  ......

  Next: NE(ifType, 18)
  # block for DS1 interfaces
  ........

}
```

UseBlock command

The UseBlock command is used to conditionally execute commands in the current block. If the condition specified is met in a block, the process continues to execute commands in the same block.

> ***Note:*** The Next and UseBlock commands both allow you to select the correct command block to handle a response. However, the conditions in these two commands have opposite effects. The Next command jumps to the next block if the condition evaluates to true. The UseBlock command continues to execute commands in the same block if the condition evaluates to true.

Syntax

```
UseBlock: <condition>
```

Parameters

`condition` A Boolean expression that indicates the condition for the current block. If this condition is met, the process continues with the commands in the same block. If this condition is not met, the process jumps to the next block.

Example

The following polling class declaration discovers device interfaces by polling the MIB-II interfaces table. The class contains several response-handling blocks that are each designed to handle a different interface type. The correct block is selected by a series of UseBlock commands that test the value of the ifType response variable.

```
CLASS ifTableConf
{
  REQGROUP 0
  COMPTYPE 3
  METHOD table

  ifIndex    1.3.6.1.2.1.2.2.1.1
  ifType     1.3.6.1.2.1.2.2.1.3
  ...

  UseBlock: EQ(ifType, 30)
  # block for DS3 interfaces
```

```
    ......

    UseBlock: EQ(ifType, 39)
    # block for SONET interfaces
    ......

    UseBlock: EQ(ifType, 18)
    # block for DS1 interfaces
    ........

}
```

Quit command

The Quit command is used to stop the processing of a response in specific conditions. When the specified condition is met in a response-handling block, the Quit command causes the system to stop processing the response at this point. For example, you can use the Quit command to discard responses that contain either of the following types of data:

- data about a table entry that is associated with a component that is not modelled
- data that cannot be handled before an associated response is processed

Syntax

```
Quit: <condition>
```

Parameters

`condition` A Boolean expression that indicates the condition for quitting. If this condition is met, processing of the current response stops.

Example

The following polling class declaration discovers device interfaces by polling the MIB-II interfaces table. Entries that describe the ATM interface are ignored because of the Quit command.

```
CLASS ifTableConf
{
  REQGROUP 0
  COMPTYPE 3
  METHOD table

  ifIndex    1.3.6.1.2.1.2.2.1.1
  ifType     1.3.6.1.2.1.2.2.1.3
  ...

  Quit: EQ(ifType, 37)
  UseBlock: EQ(ifType, 30)
  # block for DS3 interfaces
  ......

  UseBlock: EQ(ifType, 39)
  # block for SONET interfaces
  ......
```

```
  UseBlock: EQ(ifType, 18)
  # block for DS1 interfaces
  ........

}
```

Log command

The Log command is used to issue a log when an unexpected or notable situation is identified while processing a response. You can also use this command for debugging purposes, even while in the field.

Syntax

```
Log: <log level> <log contents>
```

Parameters

`log level` An expression that specifies the level at which the log is issued. Possible values are MAJOR, MINOR, ERRORS, and INFO.

`log contents` An expression that evaluates to a string specifying the contents of the issued log.

Note: Processing of this command has been optimized so that if log level is not currently selected, the log contents parameter is not evaluated.

Example

The following polling class declaration discovers device interfaces by polling the MIB-II interfaces table. The Log command is used to identify the interfaces found, or to deal with unexpected interface types.

```
CLASS ifTableConf
{
  REQGROUP 0
  COMPTYPE 3
  METHOD table

  ifIndex    1.3.6.1.2.1.2.2.1.1
  ifType     1.3.6.1.2.1.2.2.1.3
  ...

  Quit: EQ(ifType, 37)
  UseBlock: EQ(ifType, 30)
  Log: "MINOR" SPCAT("Interface" ifIndex, "is DS3")
  # block for DS3 interfaces
  ......

  UseBlock: EQ(ifType, 39)
  Log: "MINOR" SPCAT("Interface" ifIndex, "is SONET")
```

```
# block for SONET interfaces
......

UseBlock: EQ(ifType, 18)
Log: "MINOR" SPCAT("Interface" ifIndex, "is DS1")
# block for DS1 interfaces
........

Log: “ERRORS” \
     SPCAT (“Unexpected interface type:”, ifType)
```

Note: SPCAT adds a space between the concatenated strings.

Discover command

The Discover command is used to stop the processing of a response and schedule a new discovery polling cycle. This command causes the device to be rediscovered immediately instead of during the next discovery polling cycle, which may be up to 12 hours later. Immediate rediscovery is necessary if the device configuration changes, as can be indicated by the contents of some responses.

Syntax

```
Discover: <condition> [<profile name>]
```

Parameters

`condition` A Boolean expression that indicates the condition for rediscovery. If this condition is met, the data collection daemon (DCD) schedules a new discovery polling cycle and stops processing the response.

`profile name` An optional parameter that specifies a different agent profile to use in the discovery. This parameter is used to replace the current agent profile if it is not appropriate for the device. For example, a network can have several versions of devices of the same type supporting different MIBs. This option is used to switch to the correct agent profile when polling has discovered which MIB version a device supports.

Note 1: The profile name is the name specified in the agentProf option in the process global options (.cfg) file, and it is used to derive the agent profile (.agp) file name.

Note 2: If the profile name does not refer to an existing profile, the device is put in the unmanaged state. In this state, polling stops and traps are discarded.

Example

The following reachability polling request detects device name changes by polling the MIB-II sysName variable. In this example, rediscovery is performed if the current device name differs from the name derived from the contents of sysName. The MAKENAME operator processes the sysName contents to produce a device name in an acceptable format.

```
CLASS edgeReachability
{
  REQGROUP 2
  COMPTYPE 2
  METHOD single

  # polled variables
  sysName 1.3.6.1.2.1.1.5.0

  # response handling configuration
  Discover: NE(DEVNAME(), MAKENAME(sysName))
}
```

Poll command

The Poll command is used to schedule a new state polling cycle. This command causes a state polling cycle to start immediately instead of at the next scheduled state polling cycle, which may be up to five minutes later. This command is mostly intended for use in trap translation records, but it can also be used in response-handling command blocks if required.

Syntax

```
Poll: <condition> [<delay time> [<group number>
[<component type> [<instance number>]]]]
```

Parameters

`condition` A Boolean expression that indicates the condition for rescheduling. If this condition is met, the data collection daemon (DCD) schedules a new state polling cycle.

`delay time` is an integer expression that specifies the delay, in seconds, before the request is scheduled. If the value of this parameter is 0, the request is scheduled with the current time and is added to the ready request queue. If the value of this parameter is greater than 0, the current time is increased by the specified value and the request is added to the ready request queue. If this parameter is not specified, none of the following parameters can be specified.

`group number` is the identifier of the polling request group that must be scheduled on demand. The only group number that cannot be specified is the trap polling group.If this parameter is not specified, neither of the following parameters can be specified.

`component type` is an integer expression that identifies the request that must be scheduled on demand. When you specify this parameter, only the request group with the matching attribute is scheduled. If this parameter is not specified, the following parameter cannot be specified.

`instance number` is a string expression that specifies the instance to be added to each polled variable table OID to poll a single row. This value must be entered in OID format unless the table has only one integer in the index. If this parameter is not specified, the entire table is polled.

Assign command

The Assign command is used to create or update variables. The configuration data language that describes response handling provides this command to simplify the computation of required character strings or conditional expressions. The Assign command divides this computation task among several commands.

Syntax

`Assign: <variable name> <expression>`

Parameters

`variable name` The name given to the variable. The scope of the created or updated variable is the context created to handle the current response. It exists only while the current response is processed.

`expression` Any allowed expression that evaluates to a character string.

Example

The following polling class declaration discovers entries in a port table. The first Assign command is used to assign a string to the *name* local variable. The second Assign command is used to append to the local variable information computed by the previous command.

```
CLASS edgePortConf
{
  REQGROUP 0
  COMPTYPE 4
  METHOD table

  # polling variables
  board   1.3.6.1.4.1.562.12.1.1.2.1.1
  port    1.3.6.1.4.1.562.12.1.1.2.1.2
  status  1.3.6.1.4.1.562.12.1.1.2.1.6

  # response handling command block
  Assign: name [ EQ(board, 1) ? \
          SPCAT(DEVNAME(),“NWBOARD $") \
          SPCAT(DEVNAME(), "BOARD", SUBS(board, 1) ]
  Assign: name SPCAT(name, "PO", port)
  CompId: name
```

```
    Cache: SPCAT("PORT", board, port) name
    ......
}
```

The first Assign command assigns a string that depends on the value of the board polled variable. If this variable is 1, the string is obtained by the concatenation of the device name and the string constant “NWBOARD $”. If this board polled variable is not 1, the string is obtained by the concatenation of the device name, the string constant “BOARD”, and the value of the board polled variable minus 1.

The second Assign command appends to the local variable the string constant PO and port number contained in the port polled variable to the local variable. SPCAT adds a space between the concatenated strings. The *name* variable can subsequently be used to define the component identifier, create the cached entry, or perform other functions.

Compid command

Note: This command must not be confused with the Comp_id rule used in trap translation to define the component identifier attribute of the resulting alarm.

The Compid command is used to identify a component associated with a response. Subsequent commands, such as the State and Property commands, implicitly apply to the component identified by the last Compid command executed for this response.

The Compid command can also identify a device that is represented by a proxy agent. A proxy agent can collect device information and report the device's state to the manager.

The Compid command execution starts with argument validation. If the component identifier is not valid, a log is issued and the response is discarded.

The type of request determines what happens if the component does not already exist. See the table "Operations triggered by component status and request type" (page 111).

Syntax

```
Compid: <expression> [<proxy flag>]
```

Parameters

`expression` An expression that evaluates to a character string which is used as the component identifier associated with the response.

`proxy flag` An optional parameter that indicates if the device is represented by a proxy agent. Values allowed are "standard" and "proxy". The "standard" value indicates that this is a real device; "proxy" indicates that this device is represented by a proxy agent. The default is "standard".

Note: The proxy flag parameter can only be used for the device itself (top node); it cannot be used for subcomponents of the device.

Example

The following polling class declaration is a request to poll the interface table for state updates. The Compid command identifies the affected component.

```
CLASS mib2InterfState
{
  REQGROUP 1
  COMPTYPE 1
  METHOD table

  ifIndex       1.3.6.1.2.1.2.2.1.1
  ifAdminStatus 1.3.6.1.2.1.2.2.1.7
  ifOperStatus  1.3.6.1.2.1.2.2.1.8.

  Assign: name CACHE(CAT(“ifIndex”, ifIndex))
  Discover: EMPTY(name)
  Compid: name
  State: [OR(EQ(ifOperStatus, 2), \
             NE(ifAdminStatus, 1)) ? OOS \
           AND(EQ(ifOperStatus, 1), \
               EQ(ifAdminStatus, 1)) ? INSV \
               UNK]
}
```

Operations

The following table shows the operations triggered by the Compid command. These operations depend on the origin of the request and whether or not the component already exists in the response.

Table 6
Operations triggered by component status and request type

Origin of request	Status of component in response	Operations triggered by the Compid command
state and reachability groups	component does not already exist	• new discovery polling cycle is scheduled • processing of the response is terminated
	component exists	• pointer to a component is recorded for subsequent commands
discovery group (except the device discovery request)	component does not already exist	• component is created • component type and SNMP table instance properties are created
	component exists	• pointer to a component is recorded for subsequent commands

(Sheet 1 of 3)

Table 6 (Continued)
Operations triggered by component status and request type

Origin of request	Status of component in response	Operations triggered by the Compid command
device discovery	component identifier is the device name	Polling agent verification: • If another polling agent already exists for the same device name and has established device reachability, the current polling agent is deleted to stop processing of the request. Therefore the DCD avoids polling the same device through multiple sessions. • If the other polling agent has not yet established device reachability, it is deleted because the current agent has received a response from the device. If the device name of the current agent is changed, the existing components and subcomponents are deleted. They are recreated under a new name by the discovery cycle, starting with the current response. Reachability verification: • If the polling agent is marked as unreachable, the status is changed to reachable. • If the polling agent is marked as unreachable but the device component already exists, a device reachable notification is sent to the client processes. Device component creation: • If the device component does not already exist, it is created. • If the device component does not already exist, the following component properties are added: component type, IP address, community string, remote port.

(Sheet 2 of 3)

Table 6 (Continued)
Operations triggered by component status and request type

Origin of request	Status of component in response	Operations triggered by the Compid command
device discovery	component identifier is not the device name	• component is created • component type and SNMP table instance properties are created
(Sheet 3 of 3)		

State command

The State command is used to define the state of the component currently associated with the response. This command is applied only when the Compid command has already been executed for the current response instance. The current response instance does not refer to a previous response to the same request. The component state is updated if required. A state notification is sent to client processes whether or not the state has changed.

> ***Note:*** A state notification is not sent to client processes if the state was in service (INSV) and remains INSV, and no SET/CLEAR alarm was issued in the interval.

Syntax

```
State: <expression>
```

Parameters

`expression` An expression that evaluates to one of the allowed component state values. Possible values are INSV (in service), OOS (out-of-service), ISTB (in-service troubled), NEX (not existing), and UNK (unknown); see the following note. Use of the NEX value is not recommended. A better alternative for this situation is to use the Quit command before the Compid command is executed.

> ***Note:*** Consider consequences carefully before hard-coding the state of a component. In particular, it is strongly recommended not to set the state to UNK (unknown). Reachability is determined when the Compid command is issued in a polling configuration file. If the State command is issued to set the state to UNK, the reachability can be reset to unreachable for proxy devices. This situation could cause the DCD to cycle the node from INS to UNK continuously, which is a problem because the General Management Data Router (GMDR) treats any UNK node as an unmanaged node.

Example

The following polling class declaration is a request to poll the interface table for state updates. The State command defines the state of the component.

```
CLASS mib2InterfState
{
  REQGROUP 1
```

```
    COMPTYPE 1
    METHOD table

    ifIndex       1.3.6.1.2.1.2.2.1.1
    ifAdminStatus 1.3.6.1.2.1.2.2.1.7
    ifOperStatus  1.3.6.1.2.1.2.2.1.8.

    Assign: name CACHE(CAT(“ifIndex”, ifIndex))
    Discover: EMPTY(name)
    Compid: name
    State: [OR(EQ(ifOperStatus, 2), \
               NE(ifAdminStatus, 1)) ? OOS \
            AND(EQ(ifOperStatus, 1), \
                EQ(ifAdminStatus, 1)) ? INSV \
                UNK]
}
```

In this example, the device cache is searched for the component identifier associated with this value of ifIndex. The cache entry should have been created by the corresponding discovery request.

In the case of a new component, there is no cache entry; the device configuration has changed and must be rediscovered. Use the Discover command to trigger this rediscovery. Otherwise, the affected component is identified using the CompId command, and a state based on the administrative and operational status indicators is computed.

Cache command

The Cache command is used to create or update a cache entry. A cache stores information computed from responses so that it is available to process future responses or traps. The cache is an information store that is local to each polling agent or device. It is recreated by each discovery polling cycle for this device.

For example, the cache can be used to store component identifiers of subcomponents that are discovered by the discovery polling cycle. This enables the corresponding request of the state polling cycle to retrieve the identifiers instead of recomputing them. A subsequent failure of the cache search indicates that the table contains an undiscovered component.

The Cache command appears similar to the Property command, but caches differ from properties in several ways. See “Property command” (page 119) for these differences.

Syntax

```
Cache: <entry name> <entry value> [<keep flag>]
```

Parameters

`entry name` A character string that is used as a key to search the cache store.

`entry value` A character string that specifies the value that can be retrieved. If this parameter evaluates to an empty string, no cache entry is created; if a cache entry exists, it is destroyed.

`keep flag` An optional parameter that can be used to specify what happens to the cache entry when a new discovery polling cycle starts. The values allowed are “keep” and “clean”. If the parameter is absent in a response-handling command block, it defaults to “clean”.

Note: These flag values must be specified in double-quotes, for example:

```
Cache: SPCAT("PORT", board, port) name "keep"
```

Normally all the device cache entries are deleted before a discovery polling cycle starts to be recreated by the discovery requests. The keep flag parameter saves some cache entries from being destroyed before rediscovery.

Example

The following polling class declaration discovers entries in a port table. The Cache command creates a cache entry.

```
CLASS edgePortConf
{
  REQGROUP 0
  COMPTYPE 4
  METHOD table

  # polling variables
  board   1.3.6.1.4.1.562.12.1.1.2.1.1
  port    1.3.6.1.4.1.562.12.1.1.2.1.2
  status  1.3.6.1.4.1.562.12.1.1.2.1.6

  # response handling command block
  Assign: name [ EQ(board, 1) ? /
          SPCAT(DEVNAME(),“NWBOARD $") /
          SPCAT(DEVNAME(), "BOARD", SUBS(board, 1)) ]
  Assign: name SPCAT(name, "PO", port)
  CompId: name
  Cache: SPCAT("PORT", board, port) name
  ......
}
```

The first Assign command assigns a string that depends on the value of the board polled variable. If this variable is 1, the string is obtained by the concatenation of the device name and the string constant “NWBOARD $”. If this board polled variable is not 1, the string is obtained by the concatenation of the device name, the string constant “BOARD”, and the value of the board polled variable minus 1.

The second Assign command appends to the local variable the string constant PO and port number contained in the port polled variable to the local variable. SPCAT adds a space between the concatenated strings. The *name* variable can subsequently be used to define the component identifier, create the cached entry, or perform other functions.

The Cache command is used to store this component identifier against a key formed the concatenation of “PORT”, the value of board, and the value of port. The keep flag parameter is not present, so it defaults to “clean” which causes the entry to be deleted before rediscovery.

If it was necessary to keep the cache entry created when a new discovery polling cycle starts, this Cache command would have to include an explicit keep flag, for example:

```
Cache: SPCAT("PORT", board, port) name "keep"
```

Property command

The Property command is used to create or update component properties. Each component has a property list which can store configuration information discovered through polling for this component. Client processes can retrieve this information through queryProperty API requests.

The Property command appears similar to the Cache command. However, properties differ from caches in the following ways:

- Properties are stored at the component level, while cache entries are stored at the device level.
- Properties exist until the component is deleted, while cache contents are reset before each device rediscovery unless prevented by the optional keep flag parameter.
- Properties can be accessed by any client process, while cache contents can only be accessed by response-handling or trap translation commands.

Syntax

```
Property: <property name> <property value>
```

Parameters

`property name` A character string that specifies the name of the property to be created or updated.

`property value` A character string that specifies the new or updated value of the property.

Example

The standard properties are created automatically. The Property command is used in specific circumstances. For example, you can use the Property command to create properties such as version type if several versions of the same device are being supported.

Alarm command

The alarm command is used to issue an alarm as a result of polling operations. The command causes the data collection daemon (DCD) to simulate a trap sent by the device. This extends the list of traps sent by the device. A corresponding trap translation record must be configured.

For example, an alarm can be issued if a polled variable exceeds a specified threshold. This alarm has the same effect as a trap with the supplied specific code and variable list.

Syntax

```
Alarm: <condition> <trap code> <variable list>
```

Parameters

`condition` A Boolean expression that indicates the condition for the alarm. If this condition is met, the alarm is issued.

`trap_code` The specific code of the trap that identifies the translation record to use.

> ***Note:*** If the device is currently identified as an SNMPv2 device, an SNMPv2 trap is simulated. The trap notification OID is then given by the following string in the trap file:
> `enterprise_OID.0.trap_code`

`variable list` A list of trap variables. Each variable is created with a string type and an object identifier (OID) equal to 0.n. These variables can be included in the alarm comment, for example. They can also be used to generate the other alarm attributes.

Example

The following declaration is a request to send a conditional alarm that is generated by the trap translation record associated with code 1234:

```
Alarm: GT(errCount, errMax) 1234 errCount errMax
```

In this example, an alarm is sent if the errCount variable has a value higher than errMax. The errCount and errMax variables are passed as trap variables probably to be included in the alarm comment.

addAddr command

The addAddr command is used to add an address to a device's SNMP agent, for a device that can have multiple polling and trap addresses.

> ***Note:*** This command can only be used on a device that has the multiple polling or multiple trap address option enabled. For details, see multiPollAddr and multiTrapAddr in "Run-time options" (page 41).

Syntax

```
addAddr: <addr> <comm> <poll | trap>
```

Parameters

`addr` is the IP address of the device.

`comm` is the community string of the device.

`poll | trap` is the type of address, specified as either poll or trap. A polling address is an address to which polling requests can be sent, and from which traps can be received. A trap address is an address from which only traps can be received. Polling requests cannot be sent to a trap address.

Example

The following polling class declaration adds an address to a device.

```
CLASS ciscoDevSetupConf
{
  REQGROUP 0
  COMPTYPE 2500
  METHOD table

  # polling configuration
  ipAdEntAddr 1.3.6.1.2.1.4.20.1.1

  # Skip current address
  Next: EQ(ipAdEntAddr, DEVADDR())
  addAddr: ipAdEntAddr DEVCOMM() "poll"
  Log: "INFO" CAT("Adding ip address", ipAdEntAddr)
}
```

In this example, an IP address is defined by ipAdEntAddr. This variable is then used to add a polling address to the device.

clearAddr command

The clearAddr command is used to remove an address from a device's SNMP agent. An agent can have multiple polling and trap addresses.

Syntax

```
clearAddr: <addr> <comm> <poll | trap>
```

Parameters

`addr` is the IP address of the device.

`comm` is the community string of the device.

`poll | trap` is the type of address, specified as either poll or trap. A polling address is an address to which polling requests can be sent, and from which traps can be received. A trap address is an address from which traps can be received. Polling requests cannot be sent to a trap address.

> ***Note:*** You can remove all of the addresses for this device at once by using a wildcard (*) in place of the parameters for IP address, community string, and address type. If you use this option, all addresses for this device will be removed except the current active polling address.

Examples

The following declaration is a request to remove a polling address from the device agent:

```
clearAddr: 1.3.6.1.2.1.4.20.1.1 public poll
```

The following declaration is a request to remove all trap addresses from the device:

```
clearAddr: * * trap
```

NewDev command

The NewDev command causes a new device to be discovered immediately. This command is useful in a situation where one device maintains tables of addressing information for other devices. The NewDev command triggers discovery of these other devices from the polling configuration file.

Syntax

`NewDev: <addr> <comm> <profile name>`

Parameters

`addr` is the IP address of the device.

`comm` is the community string of the device.

`profile name` is a parameter that specifies the agent profile to use in the discovery. This parameter is specified in the agentProf option in the process global options (.cfg) file, and it is used to derive the agent profile (.agp) file name.

Example

The following declaration is a request to discover a new device:

```
CLASS mib2DevConf
{
  REQGROUP 0
  COMPTYPE 1
  METHOD table

  # polling configuration
  ipRouteDest .1.3.6.1.2.1.4.21.1.1

  # Skip current address
  Next: EQ(ipRouteDest, DEVADDR())
  NewDev: ipRouteDest DEVCOMM() "mib2"
  Log: "INFO" CAT ("Adding new device", ipRouteDest)
}
```

Chapter 7
Command expressions used by the SNMP Surveillance Adapter

This section describes command expressions used by the SNMP Surveillance Adapter. It contains the following topics:

See also "Operators used by the SNMP Surveillance Adapter" (page 131).

> ***Note:*** The examples in this section are specific to a particular release level of the SNMP Surveillance Adapter framework and a specific release level of a device. Actual system output can change due to framework enhancements and device updates.

Command expression types

Most of the SNMP Surveillance Adapter command arguments are expressions that are evaluated before being passed to the command. Expressions always evaluate to strings. Expressions can subsequently be converted to integers or booleans if the context requires conversion.

The maximum length of a string resulting from an expression evaluation is 1023 characters. If an expression evaluates to a longer string, it is truncated and a log is issued. Expressions can be split between several lines if required. A new line can be started wherever a space can be inserted.

The SNMP Surveillance Adapter uses four types of expressions:

Constants

Constant expressions produce the corresponding value directly. There are three constant types:

Numeric constants

Numeric constants are used to encode integer values. The syntax is a decimal integer representation which can also be negative. The range covers signed values from $-(2^{63}-1)$ to $2^{63}-1$.

The following are valid numeric constants:

```
42
0
-123456
```

The following numeric constants are **not** valid:

```
-123,456
-123 456
```

String constants

String constants are used to encode character string values. The syntax is a sequence of characters between double quotes. The following characters are supported:

- all printable characters except " and \
- \n (carriage return)
- \t (tab)

- \\ (back slash)
- \" (double quote)

The following are valid string constants:

```
"Character String"
"123,456"   (cannot be used if an integer is required)
""   (empty string)
"a"   (single character string)
"\nError:\t"   (escaped characters)
```

Note: A command line break can occur within a string constant. If more than one line is required for a command, each additional line must be introduced by a '\' at the end of the previous line. The line break can occur at any place where a blank character could occur. The first non-blank character of the continuation line is always separated by exactly one blank from the last non-blank character of the previous line, excluding the terminating '\'.

Predefined constants

For convenience, the following string values are predefined as constants:

- boolean constants, which are used in conditional expressions

```
TRUE     "1"
FALSE    "0"
```

- component state constants, which are used in the State command

```
INSV     "insv"
OOS      "oos"
ISTB     "istb"
UNK      "unk"
NEX      "nex"
```

- timestamp format constants, which are used as parameters for the TIMESTAMP operator to describe the input format; see "String operators" (page 177)

Note: Predefined constant names are not case sensitive. Two constant names with only case differences refer to the same constant.

Variables

Variables are identifiers that are associated with character strings. The scope of a variable is the set of commands used to process a single response or trap. The value of a variable extracted or computed for a given response is not accessible for subsequent responses unless it is specifically stored in the cache.

> ***Note:*** Variable names are not case sensitive. Two variable names with only case differences refer to the same variable.

There are three variable types:

- "Polled variables" (page 128)
- "Trap variables" (page 129)
- "Local variables" (page 129)

Polled variables

Polled variables are accessible in all command blocks associated with a polling class. The request declaration contains the name and object identifier (OID) of objects that are polled by this request. When the response is received, the value of each of these objects is extracted, converted to a character string, and associated with a variable by name. The associated variable has the name given to the object in the request declaration.

This process allows command expressions to access object values by using the configured polled object name, and it ensures that a value has been returned for each polled object. Otherwise, the request is considered to have failed; an ERROR log is issued and the response is discarded.

Trap variables

Trap variables are accessible in all command blocks associated with the translation of a specific trap. Each trap contains a varbind list. When a trap is received, the value of each varbind is extracted, converted to a character string, and associated with a variable that does not have a name. These variables can subsequently be

- accessed through the VAR operator; see “Context access operators” (page 132)
- given names through the Name rule; see “Name” (page 214)

Local variables

The Assign command associates a local variable name with the string obtained by evaluating an expression. This variable can then be used in expressions to access the value of this string. The value of a local variable is accessible in all command blocks only after it has been computed using the Assign command. For details, see “Assign command” (page 107).

Conditional expressions

A conditional expression is a list of <conditions, value> pairs; the optional last pair has an implicit condition with a value of TRUE. The expression returns a value associated with the first pair that has a condition evaluating to TRUE. If none of the conditions evaluate to TRUE, the empty string is returned as the value of the conditional expression.

A conditional expression has the following syntax:

```
[<pair>*]
```

where <pair> has the syntax:

```
<expression> ? <expression>
```

The value of the first expression is converted to a boolean value. 0 and " " are converted to FALSE; all other strings are converted to TRUE. This first value determines if the second expression is evaluated. If it is not, then the next pair is evaluated. Optionally, the first expression and the ’?’ can be omitted from the last pair.

The following expressions are valid conditional expressions:

```
[EQ (state, 1) ? INSV EQ (state, 2) ? OOS UNK]

[EQ (board, 1) ? SPCAT (DEVNAME (), "NWBOARD $") \
 SPCAT (DEVNAME (), "BOARD", SUBS (board, 1))]
```

Chapter 8
Operators used by the SNMP Surveillance Adapter

The SNMP Surveillance Adapter uses command expressions which include constants, variables, conditional expressions, and operators. This section describes the operators used by the SNMP Surveillance Adapter.

Most devices only require a small proportion of the many operators supported for response-handling configuration. Other operators provide rarely-used functionality that some devices may require. The SNMP Surveillance Adapter supports operators in the following categories:

- “Context access operators” (page 132)
- “Utility operators” (page 148)
- “Arithmetic operators” (page 152)
- “Relational operators” (page 158)
- “Boolean operators” (page 172)
- “Bit operators” (page 175)
- “String operators” (page 177)

See also “Operator syntax” (page 132).

> ***Note:*** The examples in this section are specific to a particular release level of the SNMP Surveillance Adapter framework and a specific release level of a device. Actual system output can change due to framework enhancements and device updates.

Operator syntax

An expression is most often in the form of an operator call, which has the following syntax:

```
<operator name>(<arg1>, <arg2>, ...)
```

where:

`arg1` is the first argument

`arg2` is the second argument

and so on.

The syntax is easy to write, parse, and extend if more operators are required. Each argument is itself an expression; this allows complex expressions to be written. To simplify overly complex expressions, use local variables to evaluate partial expressions.

Context access operators

Context access operators are used to extract information from either a polled object or the polling agent. Specifically, these operators are used to extract contextual information from

- the device associated with the current operation
- the trap being translated
- the SNMP response processed
- the device cache
- the process cache

For a table showing all of the context access operators, see “Summary of context access operators” (page 133).

For details and examples of each context access operator, see the following sections:

- “VAROID” (page 135)
- “VARTYPE” (page 136)

Summary of context access operators

The following table lists each context access operator, its required argument format, and the contents of the string it returns.

Table 7
Context access operators

Context access operator	String contents returned
VAROID(<polled var>)	OID of this polled variable
VARTYPE(<polled var>)	polled or trap variable is one of the following: integer, string, objectId, address, integer64
VARINDEX(<polled var>)	variable table index, which is the concatenation of all index values required to access this specific variable instance in the MIB table that contains it
DEVNAME()	device name currently associated with the polling agent
NODETYPE()	node type of the device, which is the first token of the device name or the Preside Multiservice Data Manager device category
DEVTYPE()	device type, which is the numeric identifier associated with the polling agent profile
DEVCOMM()	community string currently associated with the polling agent
DEVADDR()	IP address currently associated with the polling agent
DEVPORT()	remote port number currently associated with the polling agent format
DEVOID()	enterprise OID currently associated with the polling agent, which is the first OID declared using the oidFilter option in the .agp file
DEVREACH()	"1" if the polling agent is currently reachable, or "O" if it is not reachable
CACHE(<search key>)	value associated with the cache entry if found, or empty string if it is not found
PCACHE(<search key>)	value associated with the cached entry if found, or empty string if it is not found
VAR(<variable ID>)	value of the specified trap variable if found, or empty string if it is not found
VAR2(<variable ID>)	value of the specified trap variable if found, or empty string if it is not found
GENERIC()	generic code of the trap currently translated

(Sheet 1 of 2)

Table 7
Context access operators (Continued)

Context access operator	String contents returned
SPECIFIC()	specific code of the trap currently translated
VERSION()	SNMP version flag of the trap as “0” or “1”
TRAPOID()	enterprise OID for SNMPv1 trap, or notification OID for SNMPv2 trap
SYSUPTIME()	timestamp in header of SNMPv1 trap, or value of the first variable in SNMPv2 trap

(Sheet 2 of 2)

VAROID

This section describes the VAROID operator.

Syntax

VAROID(<polled var>)

Arguments

This operator requires the polled variable name. This is the name specified in this variable declaration at the beginning of the polling class declaration.

Function

This operator returns a string containing the object identifier (OID) of this variable in the SNMP response being processed.

Errors

The following errors are possible with this operator:

- the argument is not a variable name
- the variable does not exist
- the variable has no OID, which means it is not a polled variable

With all errors, an error log is issued and an empty string is returned.

Example

If the declaration of an SNMP request scanning the interface table contains the variable declaration

```
IfOperStatus   1.3.6.1.2.1.2.2.1.8
```

Then VAROID(IfOperstatus) could return the value

```
"1.3.6.1.2.1.2.2.1.8.25"
```

where “25” represents the index of the specific instance returned in the MIB table being scanned.

VARTYPE

This section describes the VARTYPE operator.

Syntax

`VARTYPE(<polled var>)`

Arguments

This operator requires the polled variable name. This is the name specified in this variable declaration at the beginning of the polling class declaration.

Function

This operator returns a string containing the type of the variable as one of “integer”, “string”, “objectid”, “address”, or “integer64”.

Errors

The following errors are possible with this operator:

- the argument is not a variable name
- the variable does not exist

With all errors, an error log is issued and an empty string is returned.

Example

If the declaration of an SNMP request scanning the interface table contains the variable declaration

```
IfOperStatus   1.3.6.1.2.1.2.2.1.8
```

Then VARTYPE(IfOperStatus) would return the value

```
"integer"
```

VARINDEX

This section describes the VARINDEX operator.

Syntax

`VARINDEX(<polled var>)`

Arguments

This operator requires the polled variable name. This is the name specified in this variable declaration at the beginning of the polling class declaration.

Function

This operator returns a string containing the variable index. This is the MIB table index of the variable instance contained in the SNMP response being processed.

Errors

The following errors are possible with this operator:

- the operator is used in a trap translation expression; there is no associated SNMP request
- the argument is not a variable name
- the variable does not exist
- the variable is not a polled variable

With all errors, an error log is issued and an empty string is returned.

Example

If the declaration of an SNMP request scanning the interface table contains the variable declaration

```
IfOperStatus   1.3.6.1.2.1.2.2.1.8
```

And if the OID associated with this variable in the response processed is

```
IfOperStatus   1.3.6.1.2.1.2.2.1.8.25
```

Then VARINDEX(IfOperStatus) would return the value

```
"25"
```

DEVNAME

This section describes the DEVNAME operator.

Syntax

```
DEVNAME()
```

Arguments

There are no arguments required.

Function

This operator returns a string containing the name currently associated with the device. This name normally contains the node type, and it may have been defined by any of the following:

- the contents of a seed file entry
- a parameter supplied to the dcdAddNode script
- information obtained from the device through polling
- the IP address and the community string extracted from a trap received from a device that has not yet been discovered

Errors

This function does not produce any error logs.

Example

If all of the following conditions are true:

- the nodeType associated with a device is XYZ
- the device has been discovered through trap-based discovery
- the device's real name has not yet been discovered through polling

Then DEVNAME() could return

```
"XYZ PUBLIC_45.130.40.12"
```

NODETYPE

This section describes the NODETYPE operator.

Syntax

```
NODETYPE()
```

Arguments

There are no arguments required.

Function

This operator returns a string containing the node type of the device, which is the component category associated at the top level with the device itself.

Errors

This function does not produce any error logs.

Example
For an iBWA 5100 base station device, NODETYPE() returns

```
"RBSE"
```

DEVTYPE
This section describes the DEVTYPE operator.

Syntax
DEVTYPE()

Arguments
There are no arguments required.

Function
This operator returns a string containing the device type, which is the number associated with the device profile used to manage the device.

Errors
This function does not produce any error logs.

Example
For a Passport 4460 device, DEVTYPE() returns

```
"6"
```

DEVCOMM
This section describes the DEVCOMM operator.

Syntax
DEVCOMM()

Arguments
There are no arguments required.

Function
This operator returns a string containing the device community string.

Errors
This function does not produce any error logs.

Example
In most cases, DEVCOMM() returns

```
"public"
```

DEVADDR
This section describes the DEVADDR operator.

Syntax
DEVADDR()

Arguments
There are no arguments required.

Function
This operator returns a string containing the IP address currently used to poll the device. This string is in a.b.c.d format.

Errors
This function does not produce any error logs.

Example
DEVADDR() could return

```
"45.135.43.17"
```

DEVPORT
This section describes the DEVPORT operator.

Syntax
DEVPORT()

Arguments
There are no arguments required.

Function
This operator returns a string containing the UDP port number of the SNMP agent on the device.

Errors
This function does not produce any error logs.

Example
In most cases, DEVPORT() returns

```
"161"
```

DEVOID
This section describes the DEVOID operator.

Syntax
DEVOID()

Arguments
There are no arguments required.

Function
This operator returns a string containing the first object identifier (OID) specified by an oidFilter option in the agent profile configuration file.

Errors
This function does not produce any error logs.

Example
For a Passport 4460 device, DEVOID() returns

```
"1.3.6.1.4.1.335.1.4"
```

This string is the sysOid of these devices.

DEVREACH
This section describes the DEVREACH operator.

Syntax
DEVREACH()

Arguments
There are no arguments required.

Function
This operator returns a string containing “1” if the device is currently responding to SNMP requests meaning that it is reachable, and “0” if the device is not responding and so is unreachable.

Errors
This function does not produce any error logs.

Example

For devices that are currently responding to polling requests, DEVREACH() returns

```
"1"
```

CACHE

This section describes the CACHE operator.

Syntax

`CACHE(<search key>)`

Arguments

This operator requires a character string containing the key of the required entry in the device cache.

Function

This operator searches the device cache for an entry with the specified key. If the entry is found, the cached value is returned; otherwise an empty string is returned.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

If a device cache contains an entry defined by the response-handling command

```
Cache: "IfIndex 25" "CIS OTT50 CARD 10 PO 4"
```

Then CACHE(“IfIndex 25”) would return

```
"CIS OTT50 CARD 10 PO 4"
```

PCACHE

This section describes the PCACHE operator.

Syntax

`PCACHE(<search key>)`

Arguments

This operator requires a character string containing the key of the required entry in the DCD process cache.

Function

This operator searches the process cache for an entry with the specified key. If the entry is found, the cached value is returned; otherwise an empty string is returned.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

If a process cache contains an entry defined by the process configuration file

```
Cache: "Addr region":"South Montreal"
```

Then PCACHE(“Addr region”) would return

```
"South Montreal"
```

VAR

This section describes the VAR operator.

Syntax

```
VAR(<variable ID>)
```

Arguments

This operator requires a character string specifying the index (starting at 0), or the object identifier (OID) of a trap variable.

Function

This operator searches the list of trap variables for the specified variable and if the variable is found, its value is converted to a string representation which is the value returned. If the variable is not found, an empty string is returned.

Note: If VAR is used for an SNMPv2 trap with an argument specifying the index of a variable, the first two default variables are skipped so that VAR(0) returns the third variable in the list. This third variable is the same as the first variable in the corresponding SNMPv1 trap.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails
- the index specified is higher than the number of variables in the list

With all errors, an error log is issued and an empty string is returned.

Example

If a generic linkDown trap is received, VAR(0) will return the contents of the first trap variable which is the IfIndex of the link interface that is going down.

VAR2

This section describes the VAR2 operator.

Syntax

```
VAR2(<variable ID>)
```

Arguments

This operator requires the character string specifying either the index (starting at 0), or the object identifier (OID) of a trap variable.

Function

This operator searches the list of trap variables for the specified variable and if the variable is found, its value is converted to a string representation which is the value returned. If the variable is not found, an empty string is returned.

Note: If VAR2 is used for an SNMPv2 trap with an argument specifying the index of a variable, the first two default variables are not skipped so that VAR2(0) returns the sysUptime. This sysUptime is always stored in the first variable of an SNMPv2 trap.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails
- the index specified is higher than the number of variables in the list

With all errors, an error log is issued and an empty string is returned.

Examples

For SNMPv2 traps only:

- VAR2(0) returns the sysUpTime variable
- VAR2(1) returns the trap notification object identifier (OID)
- VAR2(n), where n is a value greater than 1, returns the same variable as VAR(n-2)

GENERIC

This section describes the GENERIC operator.

Syntax

```
GENERIC()
```

Arguments

There are no arguments required.

Function

This operator returns a string containing the generic code of the trap currently translated. It can only be used with SNMPv1 traps. If it is used with SNMPv2 traps, an empty string is returned.

Errors

The following errors are possible with this operator:

- this function is called from the response-handling code
- the current trap is an SNMPv2 trap

With all errors, an error log is issued and an empty string is returned.

Example

If the trap currently translated is a generic linkDown trap, GENERIC() returns

```
"2"
```

SPECIFIC

This section describes the SPECIFIC operator.

Syntax

`SPECIFIC()`

Arguments

There are no arguments required.

Function

This operator returns a string containing the specific code of the trap currently translated. It can only be used with enterprise SNMPv1 traps, and it requires a generic code of 6. If the generic code is other than 6, the specific code of the trap is undefined. If this function is used with SNMPv2 traps, an empty string is returned.

Errors

The following errors are possible with this operator:

- this function is called from the response-handling code
- the current trap is an SNMPv2 trap
- the current trap is a generic trap

With all errors, an error log is issued and an empty string is returned.

Example

If the trap currently translated is a Passport 4400 systemReset trap, SPECIFIC() returns

```
"3"
```

VERSION

This section describes the VERSION operator.

Syntax

`VERSION()`

Arguments
There are no arguments required.

Function
This operator returns a string containing the SNMP version flag of the trap where:

- “0” means SNMPv1
- “1” means SNMPv2

Errors
The possible error with this operator is that the function is called from the response-handling code.

With all errors, an error log is issued and an empty string is returned.

Example
For a generic linkDowntrap, VERSION() would return

```
"0"
```

TRAPOID
This section describes the TRAPOID operator.

Syntax
```
TRAPOID()
```

Arguments
There are no arguments required.

Function
This operator returns a string containing the enterprise object identifier (OID) for an SNMPv1 trap, and the notification OID for an SNMPv2 trap.

Errors
The possible error with this operator is that the function is called from the response-handling code.

With all errors, an error log is issued and an empty string is returned.

Example

For a Shasta SNMPv1 trap, TRAPOID() would return

```
"1.3.6.1.4.1.3199.10.1"
```

SYSUPTIME

This section describes the SYSUPTIME operator.

Syntax

SYSUPTIME()

Arguments

There are no arguments required.

Function

This operator returns a string containing the timestamp contained in the header of an SNMPv1 trap, or the value of the first variable of an SNMPv2 trap. This first variable is always the value of the SysUpTime MIB variable.

Errors

The possible error is with this operator is that the function is called from the response-handling code.

With all errors, an error log is issued and an empty string is returned.

Example

For any trap, SYSUPTIME() returns a value such as

```
"1357654"
```

This value is the number of seconds since the last device restart.

Utility operators

Utility operators are used for convenience. These operators

- provide access to useful UNIX procedures
- replace more complex expressions that may be used often

For a table showing all of the utility operators, see “Summary of utility operators” (page 149).

For details and examples of each utility operator, see the following sections:

- "HOSTNAME" (page 149)
- "IFTNAME" (page 150)
- "IFTCOMP" (page 151)

Summary of utility operators

The following table lists each utility operator, its required argument format, and the contents of the string it returns.

Table 8
Utility operators

Utility operator	String contents returned
`HOSTNAME (<IP addr>)`	buffer containing the host name based on the IP address. If the host name does not exist, the string will be empty.
`IFTNAME(<num>)`	value of the expression `CAT("ifIndex", <num>)` The `IFTNAME` operator is a convenience operator used to process entries from the device interface table.
`IFTCOMP(<num>)`	value of the expression `CACHE(IFTNAME(<num>))` The `IFTCOMP` operator is a convenience operator used to process entries from the device interface table.
`BITSTR(<num>)`	value of the expression `CACHE(IFTNAME(<num>))` The `IFTCOMP` operator is a convenience operator used to process entries from the device interface table.

HOSTNAME

This section describes the HOSTNAME operator.

Syntax

```
HOSTNAME (<IP addr>)
```

Arguments
This operator requires a character string specifying the device IP address in a.b.c.d format.

Function
The network host tables are searched using the UNIX procedure *gethostbyaddr*. The corresponding host name is returned if found. Otherwise, the operator returns an empty string.

Errors
The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails
- the IP address is invalid

With all errors, an error log is issued and an empty string is returned.

Example
HOSTNAME(“45.135.23.17”) could return

```
"PARIS24"
```

IFTNAME
This section describes the IFTNAME operator.

Syntax
`IFTNAME(<num>)`

Arguments
This operator requires a character string representing an interface table index.

Function
The convenience operator used in the processing of entries from the interface table returns the value of the expression “CAT(“ifIndex”, <num>)” where:

`num` is the value of the argument supplied

Errors
The following errors are possible with this operator:

- the argument is missing

- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

IFTNAME(“25”) returns

```
"ifIndex25"
```

IFTCOMP

This section describes the IFTCOMP operator.

Syntax

IFTCOMP(<num>)

Arguments

This operator requires a character string representing an interface table index.

Function

The convenience operator used in the processing of entries from the interface table returns the value of the expression “CACHE(IFTNAME(<num>)) where:

`num` is the value of the argument supplied

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

If a device cache contains an entry defined by the response handling command

```
Cache: IFTNAME(25) "CIS OTT50 CARD 10 PO 4"
```

then IFTCOMP(25) would return

```
CIS OTT50 CARD 10 PO 4
```

Arithmetic operators

Arithmetic operators are used to perform numeric calculations on string arguments that are integer values, possibly signed, in decimal format. All calculations are done in 64-bit integer arithmetic. If conversion of one of the arguments to the corresponding integer value fails, a log is issued and the empty string is returned.

For a table showing all of the arithmetic operators, see "Summary of arithmetic operators" (page 152).

For details and examples of each arithmetic operator, see the following sections:

- "ABS" (page 153)
- "NEG" (page 154)
- "ADD" (page 154)
- "SUBS" (page 155)
- "MULT" (page 156)
- "DIVIDE" (page 157)
- "REM" (page 158)

Summary of arithmetic operators

The following table lists each arithmetic operator and the contents of the string it returns.

Table 9
Arithmetic operators

Arithmetic operator	String contents returned
`ABS(<arg>)`	absolute value of the integer value represented by the argument
`NEG(<arg>)`	product of the integer value represented by the argument and -1
`ADD(<arg1>, <arg2>)`	sum of the integer values represented by the arguments

(Sheet 1 of 2)

Table 9
Arithmetic operators (Continued)

Arithmetic operator	String contents returned
`SUBS(<arg1>, <arg2>)`	difference between the integer value represented by the first argument and the integer value represented by the second argument
`MULT(<arg1>, <arg2>)`	product of the integer value represented by the first argument and integer value represented by the second argument
`DIVIDE(<arg1>, <arg2>)`	truncated value of the division of the integer value represented by the first argument by the integer value represented by the second argument
`REM(<arg1>, <arg2>)`	remainder of the division of the integer value represented by the first argument by the integer value represented by the second argument The value of the result is defined by the expression `<arg1> - MULT(DIVIDE(<arg1>, <arg2>), <arg2>)`

(Sheet 2 of 2)

ABS

This section describes the ABS operator.

Syntax

`ABS(<arg>)`

Arguments

This argument required for this operator is a character string representing an integer value that may be signed.

Function

The operator returns a character string representing the corresponding absolute value.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails
- the argument cannot be converted to an integer value

With all errors, an error log is issued and an empty string is returned.

Example
ABS(“-12345”) returns

```
"12345"
```

NEG

This section describes the NEG operator.

Syntax
`NEG(<arg>)`

Arguments
This operator requires a character string representing a integer value that may be signed.

Function
This operator returns a character string representing the signed inverted integer value.

Errors
The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails
- the argument cannot be converted to an integer value

With all errors, an error log is issued and an empty string is returned.

Example
NEG(-12345) returns

```
"12345"
```

NEG(“2468”) returns

```
"-2468"
```

ADD

This section describes the ADD operator.

Syntax

```
ADD(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string representing an integer value that may be signed
- `arg2` is a character string representing an integer value that may be signed

Function

The operator returns a character string representing the sum of the two arguments.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- an argument cannot be converted to an integer value

With all errors, an error log is issued and an empty string is returned.

Example

ADD(“-35”, 5) returns

```
"-30"
```

SUBS

This section describes the SUBS operator.

Syntax

```
SUBS(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string representing an integer value that may be signed
- `arg2` is a character string representing an integer value that may be signed

Function
The operator returns a character string representing the subtraction of the second argument from the first.

Errors
The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- an argument cannot be converted to an integer value

With all errors, an error log is issued and an empty string is returned.

Example
SUBS(“-35”, 5) returns

```
"-40"
```

MULT

This section describes the MULT operator.

Syntax
```
MULT(<arg1>, <arg2>)
```

Arguments
This operator requires two arguments where:

- `arg1` is a character string representing an integer value that may be signed
- `arg2` is a character string representing an integer value that may be signed

Function
This operator returns a character string representing the product of the two arguments.

Errors
The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

- an argument cannot be converted to an integer value

With all errors, an error log is issued and an empty string is returned.

Example
MULT(“-35”, 5) returns

```
"-175"
```

DIVIDE

This section describes the DIVIDE operator.

Syntax
```
DIVIDE(<arg1>, <arg2>)
```

Arguments
This operator requires two arguments where:

- `arg1` is a character string representing an integer value that may be signed
- `arg2` is a character string representing an integer value that may be signed

Function
This operator returns a character string representing the truncated quotient of the first argument divided by second.

Errors
The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- an argument cannot be converted to an integer value
- the value of the second argument is 0

With all errors, an error log is issued and an empty string is returned.

Example
DIVIDE(“-35”, 3) returns

```
"-10"
```

REM

This section describes the REM operator.

Syntax

```
REM(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string representing an integer value that may be signed
- `arg2` is a character string representing an integer value that may be signed

Function

The operator returns a character string representing the integer remainder of the division of the first argument by the second.

Note: The value returned is always defined by the expression:

```
SUBS(<arg1>, MULT( DIVIDE(<arg1>, <arg2>), <arg2>))
```

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- an argument cannot be converted to an integer value
- the value of the second argument is 0

With all errors, an error log is issued and an empty string is returned.

Example

REM(“-35”, 3) returns

```
"-5"
```

Relational operators

Relational operators are used to test if their arguments meet a specific condition. If the condition is met, the value returned is 1. If the condition is not met, the value returned is 0.

For EQ, NE, GT, LT, GE, and LE operators, if both arguments represent numerical values, a numeric comparison is performed. Otherwise, lexical ordering based on ASCII character codes is tested.

The following example shows a numerical comparison:

```
EQ("001", "1")
```

A value of 1 is returned because the numerical values are equal; the condition is met. This is different from the following example:

```
EQ("12,345", 12345")
```

A value of 0 is returned because the first argument is not a numerical value and the character strings are not identical; the condition is not met.

For a table showing all of the relational operators, see “Summary of relational operators” (page 160).

For details and examples of each relational operator, see the following sections:

Summary of relational operators

The following table lists each relational operator and the condition it tests.

Table 10
Relational operators

Relational operator	Condition verified
`EQ(<arg1>, <arg2>)`	if the two arguments are equal
`NE(<arg1>, <arg2>)`	if the two arguments are not equal
`GT(<arg1>, <arg2>)`	if the first argument is greater than the second argument
`LT(<arg1>, <arg2>)`	if the first argument is less than the second argument
`GE(<arg1>, <arg2>)`	if the first argument is greater than or equal to the second argument
`LE(<arg1>, <arg2>)`	if the first argument is less than or equal to the second argument
`EMPTY(<arg>)`	if the argument is an empty string
`NUM(<arg>)`	if the argument represents a numerical value
`HEX(<arg>)`	if the argument contains only hexadecimal characters
`ALNUM(<arg>)`	if the argument contains only alphanumeric characters (letters or digits)
`LEFT(<arg1>, <arg2>)`	if the second argument is the leading part of the first argument
`RIGHT(<arg1>, <arg2>)`	if the second argument is the terminating part of the first argument
`IN(<arg1>, <arg2>)`	if the second argument is a substring of the first argument

EQ

This section describes the EQ operator.

Syntax

`EQ(<arg1>, <arg2>)`

Arguments

This operator requires two arguments where:

- `arg1` is a character string

- `arg2` is a character string

Function

The operator returns “1” if the two arguments are equal. The operator returns “0” otherwise. If both arguments can be converted to numerical values, a numerical comparison is performed. Otherwise, lexical ordering based on ASCII character codes is tested.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

EQ(“001”, 1) returns

```
"1"
```

because the numerical values are equal.

EQ(“12,345”, “12345”) returns

```
"0"
```

because the first argument is not a numerical value and the character strings are not identical.

NE

This section describes the NE operator.

Syntax

```
NE(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string
- `arg2` is a character string

Function

This operator returns “1” if the two arguments are not equal. Otherwise, the operator returns “0”. If both arguments can be converted to numerical values, a numerical comparison is performed. Otherwise, lexical ordering based on ASCII character codes is tested.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

NE(“001”, 1) returns

```
"0"
```

because the numerical values are equal.

NE(“12,345”, “12345”) returns

```
"1"
```

because the first argument is not a numerical value and the character strings are not identical.

GT

This section describes the GT operator.

Syntax

```
GT(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string
- `arg2` is a character string

Function

This operator returns “1” if the first argument is greater than the second. The operator returns “0” otherwise. If both arguments can be converted to numerical values, a numerical comparison is performed. Otherwise, lexical ordering based on ASCII character codes is tested.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

GT(“001”, -5) returns

```
"1"
```

because the first numerical value is greater than the second.

GT(“12,345”, “12345”) returns

```
"0"
```

because the first argument is not numerical and the ASCII code of ‘,’ (44) is less than the ASCII code of ‘3’ (51).

LT

This section describes the LT operator.

Syntax

```
LT(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string
- `arg2` is a character string

Function

This operator returns “1” if the first argument is less than the second. The operator returns “0” otherwise. If both arguments can be converted to numerical values, a numerical comparison is performed. Otherwise, lexical ordering based on ASCII character codes is tested.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

LT(“001” -5) returns

```
"0"
```

because the first numerical value is not less than the second.

LT(“12,345”, “12345”) returns

```
"1"
```

because the first argument is not numerical and the ASCII code of ‘,’ (44) is less than the ASCII code of ‘3’ (51).

GE

This section describes the GE operator.

Syntax

```
GE(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string
- `arg2` is a character string

Function

This operator returns “1” if the first argument is greater than or equal to the second. The operator returns “0” otherwise. If both arguments can be converted to numerical values, a numerical comparison is performed. Otherwise, lexical ordering based on ASCII character codes is tested.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

GE(“001”, -5) returns

```
"1"
```

because the first numerical value is greater than the second.

GE(“12,345”, “12345”) returns

```
"0"
```

because the first argument is not numerical and the ASCII code of ‘,’ (44) is less than the ASCII code of ‘3’ (51).

LE

This section describes the LE operator.

Syntax

```
LE(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string
- `arg2` is a character string

Function

This operator returns “1” if the first argument is less than or equal to the second. The operator returns “0” otherwise. If both arguments can be converted to numerical values, a numerical comparison is performed. Otherwise, lexical ordering based on ASCII character codes is tested.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

LE(“001”, -5) returns

```
"0"
```

because the first numerical value is greater than the second.

LE(“12,345”, “12345”) returns

```
"1"
```

because the first argument is numerical and the ASCII code of ‘,’ (44) is less than the ASCII code of ‘3’ (51).

EMPTY

This section describes the EMPTY operator.

Syntax

```
EMPTY(<arg>)
```

Arguments

This operator requires a character string.

Function

This operator returns “1” if the argument is an empty string. Otherwise, the operator returns “0”.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

EMPTY(“ABCD”) returns

```
"0"
```

because the argument is not an empty string.

EMPTY(“”) returns

```
"1"
```

NUM

This section describes the NUM operator.

Syntax

```
NUM(<arg>)
```

Arguments

This operator requires a character string.

Function

This operator returns “1” if the argument can be converted to an integer value. Otherwise, the operator returns “0”.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

NUM(“-1234”) returns

```
"1"
```

NUM(“12,345”) returns

```
"0"
```

HEX

This section describes the HEX operator.

Syntax

```
HEX(<arg>)
```

Arguments

This operator requires a character string.

Function

This operator returns “1” if the argument contains only hexadecimal characters. Otherwise, the operator returns “0”.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

HEX(“AaBbCcD”) returns

```
"1"
```

HEX(“AB-CD”) returns

```
"0"
```

ALNUM

This section describes the ALNUM operator.

Syntax

```
ALNUM(<arg>)
```

Arguments

This operator requires a character string.

Function

This operator returns “1” if the argument contains only alpha-numerical characters. Otherwise, the operator returns “0”.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

ALNUM(“123ABcd”) returns

```
"1"
```

ALNUM(“12,345”)
The comma here causes the operator to return

```
"0"
```

ALNUM(“ ABCD”)
The leading space here causes the operator to return

```
"0"
```

LEFT

This section describes the LEFT operator.

Syntax

```
LEFT(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string
- `arg2` is a character string

Function

This operator returns “1” if the second argument forms the leading part of the first.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

LEFT(“ABCDE”, “AB”) returns

```
"1"
```

LEFT(“ABCDE”, “ A”)
The leading space in the second argument causes the operator to return

```
"0"
```

RIGHT

This section describes the RIGHT operator.

Syntax

```
RIGHT(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string
- `arg2` is a character string

Function

This operator returns “1” if the second argument forms the last part of the first.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

RIGHT(“ABCDE”, “DE”) returns

```
"1"
```

RIGHT(“ABCDE”, “de”) returns

```
"0"
```

IN

This section describes the IN operator.

Syntax

```
IN(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string
- `arg2` is a character string

Function

This operator returns “1” if the second argument forms a substring of the first.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

IN(“ABCDE”, “BC”) returns

```
"1"
```

IN(“ABCDE”, “ A”)
The leading space in the second argument that is not present in the first argument causes the operator to return

```
"0"
```

Boolean operators

Boolean operators are used to combine Boolean values in order to produce another Boolean value. Although all arguments are strings, or converted to strings, they are first converted to Boolean values before the operator is applied. "0" and the empty string are converted to FALSE. All other strings are converted to TRUE.

For a table showing all of the Boolean operators, see “Summary of Boolean operators” (page 172).

For details and examples of each Boolean operator, see the following sections:

- “NOT” (page 173)
- “AND” (page 173)
- “OR” (page 174)

Summary of Boolean operators

The following table lists each Boolean operator and the value returned for different arguments.

Table 11
Boolean operators

Boolean operator	Value returned
`NOT(<arg>)`	TRUE if its argument evaluates to FALSE FALSE if its argument evaluates to TRUE
`AND(<arg1>, ...)`	TRUE if all the arguments evaluate to TRUE FALSE if any arguments evaluate to FALSE This operator accepts a variable number of arguments.
`OR(<arg1>, ...)`	TRUE if at least one of the arguments evaluates to TRUE FALSE if all of the arguments evaluate to FALSE This operator accepts a variable number of arguments.

NOT

This section describes the NOT operator.

Syntax

```
NOT(<arg>)
```

Arguments

This operator requires a character string.

Function

This operator returns TRUE if the argument is FALSE. Otherwise, the operator returns FALSE.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

NOT(IN(“ABCD”, “bc”)) returns

```
TRUE
```

NOT(EMPTY(“ABCD”)) returns

```
TRUE
```

AND

This section describes the AND operator.

Syntax

```
AND(<arg1>, ...)
```

Arguments

This operator accepts a variable number of arguments which are all character strings converted to Boolean values.

Function

This operator returns TRUE if all of the arguments evaluate to TRUE. Otherwise, the operator returns FALSE.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

AND(IN(“ABCDEFG”, “BC”), RIGHT(“ABCDEFG”, “FG”)) returns

```
TRUE
```

OR

This section describes the OR operator.

Syntax

```
OR(<arg1>, ...)
```

Arguments

This operator accepts a variable number of arguments which are all character strings converted to Boolean values.

Function

This operator returns TRUE if one of the arguments evaluates to TRUE. Otherwise, the operator returns FALSE.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

OR(IN(“ABCDEFG”, “BC”), IN(“ABCDEFG”, “BB)) returns

```
TRUE
```

Bit operators

Bit operators are used to perform bit-wise operations on integer values corresponding to string arguments that are integer values in decimal form. Bit operations are applied to 64-bit integer values. If conversion of an argument to the corresponding integer value fails, a log is issued and the empty string is returned.

For a table showing all of the bit operators, see “Summary of bit operators” (page 175).

For details and examples of each bit operator, see the following sections:

- “BITAND” (page 175)
- “BITOR” (page 176)

Summary of bit operators

The following table lists each bit operator and the string it returns.

Table 12
Bit operators

Bit operator	String returned
`BITAND(<arg1>, <arg2>)`	value obtained by *AND*ing each bit of integer value corresponding to <arg1> with the corresponding bit of the integer value corresponding to <arg2>
`BITOR(<arg1>, <arg2>)`	value obtained by *OR*ing each bit of integer value corresponding to <arg1> with the corresponding bit of the integer value corresponding to <arg2>

BITAND

This section describes the BITAND operator.

Syntax

```
BITAND(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string representing an integer value that may be signed
- `arg2` is a character string representing an integer value that may be signed

Function

This operator returns a character string representing the integer value obtained by ANDing each bit of the integer value corresponding to the first argument with the matching bit of the integer value corresponding to the second argument.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- an argument cannot be converted to an integer value

With all errors, an error log is issued and an empty string is returned.

Example

BITAND(“10”, “1”) returns

```
"0"
```

BITAND(“10”, “3”) returns

```
"2"
```

BITOR

This section describes the BITOR operator.

Syntax

```
BITOR(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is a character string representing an integer value that may be signed
- `arg2` is a character string representing an integer value that may be signed

Function

This operator returns a character string representing the integer value obtained by ORing each bit of the integer value corresponding to the first argument with the matching bit of the integer value corresponding to the second argument.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- an argument cannot be converted to an integer value

With all errors, an error log is issued and an empty string is returned.

Example

BITOR(“10”, “1”) returns

```
"11"
```

BITOR(“10”, “3”) also returns

```
"11"
```

String operators

String operators are used for string manipulation; their arguments are never modified. The result is a newly-allocated string value.

For a table showing all of the string operators, see “Summary of string operators” (page 178).

For details and examples of each string operator, see the following sections:

- “CAT” (page 179)

Summary of string operators

The following table lists each string operator and the string it returns.

Table 13
String operators

String operator	String returned
`CAT(<arg1>, <arg2>, ...)`	concatenation of the arguments
`SPCAT(<arg1>, <arg2>, ...)`	concatenation of the arguments with space character inserted between each consecutive argument pair
`LENGTH(<arg>)`	length of the string defined by its argument
`TRIM(<arg>)`	copy of its argument without leading and trailing spaces

(Sheet 1 of 2)

Table 13
String operators (Continued)

String operator	String returned
`UPPER(<arg>)`	copy of its argument in which each lowercase letter has been replaced by the corresponding uppercase letter
`LOWER(<arg>)`	copy of its argument in which each uppercase letter has been replaced by the corresponding lowercase letter
`OIDTOSTR(<table index string>)`	character string that corresponds to the table instance index string supplied
`MAKENAME(<arg>)`	valid device name converted from the supplied name
`REPLACE(<arg1>, <arg2>, <arg3>, <start>, <count>)`	string replacement based on the specified substring
`TRANSLATE(<arg1>, <arg2>, <arg3>)`	string replacement based on the specified characters
`TOKEN(<arg1>, <arg2>, <start>, <count>)`	substring extraction based on the specified tokens
`MATCH(<arg1>, <arg2>, <arg3>)`	substring extraction based on the matching pattern
`SUBSTRING(<arg1>, <arg2>, <arg3>)`	substring extraction based on the specified positions
`TIMESTAMP(<arg1>, <arg2>)`	a timestamp in the standard Preside Multiservice Data Manager format
`BITSTR(<arg1>, <arg2>)`	a binary string replacement based on the specified octet string
`HEXSTR(<arg1>, <arg2>)`	a hexidecimal value converted from the specified integer value

(Sheet 2 of 2)

CAT

This section describes the CAT operator.

Syntax

```
CAT(<arg1>, <arg2>, ...)
```

Arguments
This operator accepts a variable number of arguments which are all character strings.

Function
This operator returns a string formed by the concatenation of all the arguments.

Errors
The possible error with this operator is that the evaluation of an argument fails.

With all errors, an error log is issued and an empty string is returned.

Example
CAT(“AB”, “”, “CD”, “ EF”) returns

```
"ABCD EF"
```

In this string returned, there is no space between the first argument (AB) and the third argument (CD) because the second argument is an empty string. Note there is a space at the start of the fourth argument (EF). This space is part of the original fourth argument and is maintained throughout the concatenation.

SPCAT
This section describes the SPCAT operator.

Syntax
```
SPCAT(<arg1>, <arg2>, ...)
```

Arguments
This operator accepts a variable number of arguments which are all character strings.

Function
The operator returns a string formed by the concatenation of the arguments with a space inserted between each argument pair.

Note: If an argument is an empty string, no space is inserted before it.

Errors

The possible error with this operator is that the evaluation of an argument fails.

With all errors, an error log is issued and an empty string is returned.

Example

SPCAT(“AB”, “”, “CD”, “ EF”) returns

```
"AB CD  EF"
```

In this string returned, the second argument is ignored because it is an empty string. Therefore there is one space between the first argument (AB) and the third argument (CD). Note there are two spaces between the third and fourth arguments. One of these spaces is the space added between arguments. The other space is at the start of the fourth argument (EF). This space is part of the original fourth argument and is maintained throughout the concatenation.

LENGTH

This section describes the LENGTH operator.

Syntax

LENGTH(<arg>)

Arguments

This operator requires a character string.

Function

This operator returns a string representing an integer value that is the length of the argument, including leading and trailing spaces.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example
LENGTH(“ABDC”) returns

```
"4"
```

TRIM

This section describes the TRIM operator.

Syntax
`TRIM(<arg>)`

Arguments
This operator requires a character string.

Function
This operator returns a string obtained from its argument by removing leading and trailing white space characters (space, tab, carriage-return, new line, vertical tab, and form-feed).

Errors
The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example
TRIM(“ AB CD “) returns

```
"AB CD"
```

In this string returned, there are no leading or trailing spaces; they have been trimmed from the original string. Note the space between the AB and CD is contained within the original string and is maintained throughout the trimming operation.

UPPER

This section describes the UPPER operator.

Syntax
`UPPER(<arg>)`

Arguments
This operator requires a character string.

Function
This operator returns a string obtained from its argument by replacing each lowercase letter with the corresponding uppercase letter.

Errors
The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example
UPPER(“123abc456DE”) returns

```
"123ABC456DE"
```

LOWER
This section describes the LOWER operator.

Syntax
```
LOWER(<arg>)
```

Arguments
This operator requires a character string.

Function
This operator returns a string obtained from its argument by replacing each uppercase letter with the corresponding lowercase letter.

Errors
The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

LOWER(“123abc456DE”) returns

```
"123abc456de"
```

OIDTOSTR

This section describes the OIDTOSTR operator.

Syntax

`OIDTOSTR(<table index string>)`

Arguments

This operator requires a character string representing part of a table element object identifier (OID).

Function

This operator returns a character string obtained by translating the argument. The argument should represent a string-valued MIB table index. The OID segment corresponding to such a table index has the following form:

```
[<length>].<ascii code 1>.<ascii code 2>.......
<ascii code n>
```

The initial element is a length indication, which is optional. This operator expects an argument in this format and returns the corresponding character string.

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

OIDTOSTR(“6.78.111.114.116.101.108”) returns

```
"Nortel"
```

MAKENAME

This section describes the MAKENAME operator.

Syntax

MAKENAME(<arg>)

Arguments

This operator requires a character string.

Function

This operator returns a valid device name derived from its argument by

- removing leading and trailing white space characters
- changing all letters to uppercase
- replacing all characters which are not letters, digits, dashes (-), underscores (_) or periods (.) by underscores (_)
- truncating the string to 16 characters
- pre-pending the node type followed by a space

Errors

The following errors are possible with this operator:

- the argument is missing
- the evaluation of the argument fails

With all errors, an error log is issued and an empty string is returned.

Example

An iBWA 5100 base station MAKENAME(“ Abcd 123+def=4567 “) returns

```
"RBSE ABCD_123_DEF_456"
```

REPLACE

This section describes the REPLACE operator.

Syntax

REPLACE(<arg1>, <arg2>, <arg3>, <start>, <count>)

Arguments

The operator requires five arguments where:

- `arg1` is the source character string
- `arg2` is the character string that will be replaced

- `arg3` is the character string that will be substituted
- `start` is a character string translated to a positive integer value specifying the first disjoint occurrence to replace
- `count` is a character string translated to an integer value specifying how many occurrences to replace (-1 means all the disjointed occurrences from the occurrence specified by the fourth parameter to the end of the string)

Function

This operator returns a character string derived from the first argument in which some disjointed occurrences of the character string defined by the second argument have been replaced by the character string defined by the third argument. The fourth and fifth arguments specify which occurrences are replaced.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- the second argument is an empty string
- the fourth or fifth argument is not a valid integer value

With all errors, an error log is issued and an empty string is returned.

Example

REPLACE(“abababababababababaa”, “baba”, “X”, 2, 1) returns

```
"ababaXbabababababa"
```

REPLACE(“abababababababababaa”, “baba”, “X”, 1,2) returns

```
"aXXbabababababa"
```

REPLACE(“abababababababababaa”, “baba”, “X”,2, -1) returns

```
"ababaXXXX"
```

TRANSLATE

This section describes the TRANSLATE operator.

Syntax

```
TRANSLATE(<arg1>, <arg2>, <arg3>)
```

Arguments

This operator requires three arguments where:

- `arg1` is the source character string
- `arg2` is the character string containing the characters to replace
- `arg3` is the character string containing the characters to substitute

Function

This operator returns a character string derived from the first argument by replacing each character occurring in the second argument by the character at the corresponding position in the third argument.

Note 1: Characters from the source string are translated from left to right.

Note 2: If the second argument is longer than the third, characters in the second argument with no corresponding characters in the third argument are deleted.

Note 3: If the second argument is shorter than the third, characters in the third argument with no corresponding character in the second are ignored.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails

With all errors, an error log is issued and an empty string is returned.

Example

TRANSLATE(“SSG/OTT30 CARD/1 PO/5”, “/”, “ “) replaces slashes (/) by spaces and returns

```
"SSG OTT30 CARD 1 PO 5"
```

TOKEN

This section describes the TOKEN operator.

Syntax

```
TOKEN(<arg1>, <arg2>, <start>, <count>)
```

Arguments

The operator requires four arguments where:

- `arg1` is the source character string
- `arg2` is the character string containing the token delimiters
- `start` is a character string translated to a positive integer value specifying the first token to extract
- `count` is a character string translated to an integer value specifying how many tokens to extract (-1 means all the tokens from the token specified by the third parameter to the end of the string)

Function

This operator returns a character string containing one or many tokens extracted from the first argument. Tokens are defined as substrings separated by one or many delimiters defined by the second argument. The tokens to extract are specified by the third and fourth arguments.

> ***Note:*** If many tokens are extracted, the delimiters separating them are also included in the result.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- the second argument is an empty string
- the third or fourth argument is not a valid integer value

With all errors, an error log is issued and an empty string is returned.

Example

TOKEN(“ SSG OTT30 CARD 1 PO 5”, “ “, 3, 2) returns

```
"CARD 1"
```

TOKEN(“1.3.6.1.4.1.512.1.1.3.5”, “.”, 10, -1) returns

```
"3.5"
```

This could be an index in a MIB table.

MATCH

This section describes the MATCH operator.

Syntax

```
MATCH(<arg1>, <arg2>, <arg3>)
```

Arguments

This operator requires three arguments where:

- `arg1` is the source character string
- `arg2` is a character string representing a pattern. This pattern must contain at least one brace which is a subpattern used to extract a substring. A brace is bracketed by **“\\(“** and **“\\)”** (note the double **‘\’** required to introduce a single back-slash).

  Note: The MATCH operator supports *Basic Regular Expressions* as documented by the UNIX man pages (**man -s5 regex**).

- `arg3` is a character string translated to an integer value specifying which brace is used to extract the result (brace counting starts at 0)

Function

The pattern is applied to the source string. The operator returns the substring matching the brace specified by the third argument.

Errors

The following errors are possible with this operator:

- an argument is missing
- the evaluation of an argument fails
- the second argument is an empty string
- the second argument is not a valid pattern
- the third argument is not a valid integer value
- the third argument specifies a brace that does not exist in the pattern

With all errors, an error log is issued and an empty string is returned.

Example

MATCH(“SSG OTT30 CARD 1 PO 5”,”\\(CARD [0-9]*\\)”, 0) returns

```
"CARD 1"
```

SUBSTRING

This section describes the SUBSTRING operator.

Syntax

`SUBSTRING(<arg1>, <arg2>, <arg3>)`

Arguments

This operator requires three arguments where:

- `arg1` is the source character string
- `arg2` is a character string translated to an integer value specifying a character index, starting at 0, either counted from the beginning or the end of the first argument
- `arg3` is a character string translated to an integer value specifying how many characters to extract. If this argument produces a negative value, extraction is from the end of the first argument.

Function

This operator extracts from the first argument the substring specified by the second and third arguments.

Errors

The following errors are possible with this operator:

- an argument is missing.
- the evaluation of an argument fails
- the second or third argument is not a valid integer value

With all errors, an error log is issued and an empty string is returned.

Example

SUBSTRING(str, 0, 3) returns the first three characters of *str*.

SUBSTRING(str, 0, -3) returns the last three characters of *str*.

SUBSTRING(str, 2, 5) returns five characters from *str* starting at the third (index = 2).

SUBSTRING(str, 3, 9999) returns all but the first three characters of *str*.

SUBSTRING(str, 3, -9999) returns all but the last three characters of *str*.

TIMESTAMP

This section describes the TIMESTAMP operator.

Syntax

```
TIMESTAMP(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is the character string containing the source timestamp
- `arg2` is a character string identifying the format of the first argument (named constants have been defined for each of the supported formats)

Function

The timestamp contained in the first argument is converted in a timestamp suitable for inclusion into an Preside Multiservice Data Manager alarm.

Errors

The following errors are possible with this operator:

- an argument is missing.
- the evaluation of an argument fails
- the first argument is an empty string
- the format specified by the second argument is not supported
- the first argument format does not match the second argument

With all errors, an error log is issued and an empty string is returned.

Example

TIMESTAMP(“Tue Sep 01 08:17:52 1992”, UNIXFORMAT) returns

```
"1992 09 01 08 17 52"
```

TIMESTAMP(“TUE SEP 01 08:17:52 1992”, UPPERUNIXFORMAT) also returns

```
"1992 09 01 08 17 52"
```

TIMESTAMP(“1992-09-01 08:17:52”, SEPFORMAT) also returns

```
"1992 09 01 08 17 52"
```

TIMESTAMP(“08:17:52”, TIMEFORMAT) could return

```
"2001 01 24 08 17 52"
```

where today’s date has been added to the time supplied by the first argument.

The following formats are also supported:

- EPOCHFORMAT in which the first argument must represent an integer value that is the number of seconds since 1970-01-01 00:00:00
- SNMPV2FORMAT in which the first argument is converted to a data structure representing a timestamp according to an SNMPv2 convention

BITSTR

This section describes the BITSTR operator.

Syntax

`BITSTR(<arg1>, <arg2>)`

Arguments

This operator requires two arguments where:

- `arg1` is the expression that will be converted
- `arg2` is an expression identifying the format of the first argument. Two formats are supported: “octetstring” and “positive”.
  - — If the format is “octetstring”, the value of the first argument will be treated as an octet string.
  - — If the format is “positive”, the value of the first argument will be treated as a positive decimal integer.

Function

The octet string or the positive integer contained in the first argument is converted to a binary string.

Errors

The following errors are possible with this operator:

- the evaluation of an argument fails because the first argument cannot be evaluated as a positive integer value and the value of the second argument is “positive”
- the format specified by the second argument is not supported

With all errors, an error log is issued and an empty string is returned.

Example

BITSTR(strvar, “octetstring”) returns

```
"0000010101110000"
```

if strvar is a polled variable that is in octet string format with a value of x05 70

BITSTR(strvar, “positive”) returns

```
"00000000000000000000000000000000000000000000000000000000100001"
```

if strvar is a polled variable that is a positive integer with a value of 17

Limitations

This operator has a length limit of 127 octets. Octets after the 127th octet will not be converted.

If an octet string contains NULL octets, the first NULL octet and all following octets will be discarded in the returned value of all operators except BITSTR. To have all NULL octets converted, you must use one for the following variables, for the first argument of the BITSTR operator:

- a polled variable
- a variable in a trap PDU after the “Name” rule
- one of the VAR/VAR2 operators in the trap translation file

- a variable that is assigned an expression that is composed of a polled variable, a variable that follows the “Name” rule in a trap PDU, or one of the VAR/VAR2 operators in the trap translation file

HEXSTR

This section describes the HEXSTR operator.

Syntax

```
HEXSTR(<arg1>, <arg2>)
```

Arguments

This operator requires two arguments where:

- `arg1` is the expression that will be converted
- `arg2` is an expression identifying the format of the first argument. Two formats are supported: “octetstring” and “positive”.
  - If the format is “octetstring”, the value of the first argument will be treated as an octet string.
  - If the format is “positive”, the value of the first argument will be treated as a positive decimal integer.

Function

The expression contained in the first argument is converted to a hexadecimal string.

Errors

The following errors are possible with this operator:

- the evaluation of an argument fails because the first argument cannot be evaluated as a positive integer value and the value of the second argument is “positive”
- the format specified by the second argument is not supported

With all errors, an error log is issued and an empty string is returned.

Example

HEXSTR(strvar, “octetstring”) returns

```
"0570"
```

if strvar is a polled variable or a trap's varbind that is in octet string format with a value of x05 70

HEXSTR(strvar, "positive") returns

```
"0000000000000001F"
```

if strvar is a polled variable or a trap's varbind that is a positive integer with a value of 31

Chapter 9
Trap translation configuration

This section describes the rules for trap translation. It contains the following topics:

Trap translation file

The trap translation file is a configuration file. It consists of several trap translation records. Each record is separated from the next record by a blank line.

The trap translation configuration file is located in one of the following directories:

1 /opt/MagellanNMS/cfg/dcd/
This is the directory where customers should create their files, or store modified copies of default files.

2 /opt/MagellanNMS/ext/lib/cfg/dcd/
This is the directory where other Nortel Networks packages should install their configuration files.

3 /opt/MagellanNMS/lib/cfg/dcd/
This is the directory for Preside Multiservice Data Manager (MDM) installation from the CD-ROM.

The search priority for the trap translation file is as ordered in this list. If a required trap translation file is not in the customer directory location, the directory for other Nortel Networks packages is searched next, and the MDM directory is searched last.

This means that customer changes take priority. A customer file is used even if a default file with the same name exists.

CAUTION
Risk of losing default file updates
If a customer defines or modifies a file, it is the customer's responsibility to update this file when new loads are installed. The customer file must be updated with any new material present in a newly installed version of the default file.

For more information about configuration files for the SNMP Surveillance Adapter, see "Configuration files" (page 37).

Directives

The trap translation configuration file (.tra) can contain include directives. Include directives instructs the compiler to include trap translation records that are contained in another file. They have the following syntax:

```
#include <file name>
```

where:

`file name` identifies the file to be included. If `<file name>` does not include the directory path, a search priority is used. If a directory path is included, the search priority is not used. For more information about the search priority, see “Trap translation file” (page 197).

Included files can also contain include directives:

- Files included in .tra files using this directive can only contain complete translation records.
- Include directives must precede all trap translations records declared in the file.

They can be used to ensure that several device profiles translate the same trap into the same alarm. For example, the following files can be included in any .tra file to provide a common way to handle generic traps:

- genRestart.tra for coldStart and warmStart traps
- genLink.tra for linkUp and linkDown traps
- genAuth.tra for authenticationFailure traps

These files can be found in /opt/MagellanNMS/lib/cfg/dcd.

Translation records

Translation records are required in the trap translation configuration file. This section describes the following:

Function of translation records

Trap translation records contain information that the SNMP Surveillance Adapter uses to translate traps into Preside Multiservice Data Manager alarms. Translation records allow the trap translation process to be fully data-driven.

Each record specifies the rules to translate one type of trap. The type of trap is identified by object identifiers (OIDs) and codes. A default translation record can be provided to translate traps for which no translation record is defined.

Format of translation records

A trap translation record is a group of lines separated from the next record by a blank line. Each record has the following format:

- Trap_oid. The first line associates the record with one of the following:
  - the default translation
  - an enterprise OID for SNMPv1 traps
  - a notification OID for SNMPv2 traps
- Trap_code. The second line specifies the associated trap code using one of the following:
  - a generic code for SNMPv1 generic traps
  - a specific code for SNMPv1 enterprise traps
  - the string “v2” for SNMPv2 traps or INFORMS
- Rule label and expression. Each subsequent line specifies a rule used for one of the following functions:
  - obtaining one of the alarm attributes
  - performing flow control
  - defining variables
  - performing special actions

Example of a translation record

The following is an example of a translation record:

```
Trap_oid: 1.3.6.1.2.1
Trap_code: -1, -2
Comp_id: E DEVNAME()
Fault_code: E [EQ(GENERIC(),0) ? "C0000000" \
                  "C0000001"]
Severity: S cleared
Event: S clear
```

```
AlmType: S operator
ProbCause: S operationalConditions
Comment: E [EQ(GENERIC(),0) ? "Cold start trap" \
            "Warm start trap"]
ClrScope: S clearHier
Discover: E TRUE
```

For an explanation of this example, see “Expression examples” (page 227).

Trap identification

The following rules are used to identify traps:

- “Trap_oid rule” (page 201)
- “Trap_code rule” (page 202)

Trap_oid rule

This rule is the first line of each translation record. It is used to associate the record either with the default translation or with an OID. The trap_oid rule has the following format:

```
Trap_oid: DEFAULT | <notification OID> | <enterprise
          OID>
```

where:

`DEFAULT` identifies a record used to translate a trap for which there is no translation record defined. This record is useful as a catch-all for new traps showing up in a new device version.

`notification OID` is an identification associated with translation records for SNMPv2 traps.

`enterprise OID` is an identification associated with translation records for SNMPv1 traps. This can be replaced by the keyword `ALL`, which causes the translation record to be used by default for any enterprise OID. Using wildcards for generic traps allows a single trap translation configuration file to be used for all device types supported at the MIB-II level.

Note: A translation record using the keyword ALL has lower priority than another record with a specified enterprise OID for traps carrying this OID.

Example

This example defines two trap translation records for cold start traps:

```
Trap_oid: ALL
Trap_code: -1
........

Trap_oid: 1.3.6.1.4.1.562.12
Trap_code: -1
........
```

All cold start traps, except those carrying an enterprise OID of 1.3.6.1.4.1.562.12, will use the first translation record.

Trap_code rule

This rule is the second line of each translation record. For SNMPv1 traps, it is used to specify the associated trap code. For SNMPv2 traps, this rule only indicates that the record is designed to handle SNMPv2 traps which cannot have trap codes. The trap_code rule has the following format:

```
Trap_code: <code>[, <code>, <code>, ...] | V2
```

where:

`code` is an integer value that specifies the associated trap code as follows:

- a positive or null value is associated with the specific code of an SNMPv1 enterprise trap
- a negative value is associated with an SNMPv1 generic trap by the relation
  `<trap code> = -(<generic code> + 1)`
  This avoids ambiguity between the generic code 0 and a specific code 0, which are both valid.

The Trap_code rule can contain a comma-separated list of trap codes. This avoids the duplication of almost identical translation records for different but similar traps.

For SNMPv1 traps, the GENERIC and SPECIFIC functions can be used in the translation record to generate specific code values for an attribute if required. For details, see “Functions used in translation records” (page 229).

Examples

These first two examples specify multiple associated trap codes. The third example is for SNMPv2 traps.

```
Trap_code: -3, -4

Trap_code: 6, 7, 8, 11, 12

Trap_code: V2
```

Rule labels

A rule label identifies the function performed by the rule. Each of these translation record lines has the following syntax:

```
<rule label>: <E-expres> | <S-expres> | <V-expres> |
              <C-expres> | <P-expres>
```

For information about the expression formats, see “Expressions used in translation records” (page 223).

The following categories of rule labels are supported:

- “Alarm attributes” (page 203)
- “Flow control” (page 213)
- “Variable definition” (page 214)
- “Special actions” (page 215)

Alarm attributes

The following rule labels identify the alarm attribute generated:

- “Comp_id” (page 204)
- “Alm_time” (page 204)
- “Fault_code” (page 205)
- “Event” (page 206)
- “Severity” (page 206)
- “Comment” (page 207)
- “AlmType” (page 208)
- “ProbCause” (page 208)

Comp_id

The Comp_id rule produces the alarm component identifier in internal Preside Multiservice Data Manager (MDM) format. The MDM format is a list of component (<category>, <instance value>) pairs that uniquely identify the component in the network. Each pair is separated from the next by a blank character. The category is also separated from the instance value by a blank character.

This attribute is mandatory. If it is not defined by the translation record, the trap is discarded.

> ***Note:*** This rule must not be confused with the Compid command used in response-handling to identify a component associated with a response.

Example

```
Comp_id: E SPCAT (DEVNAME(), "PO", index)
```

This rule uses the E-expression to create the alarm component identifier from the discovered device name (returned by DEVNAME()), "PO" category, and port index. SPCAT adds a space between the concatenated strings.

Alm_time

The Alm_time produces the alarm timestamp in Preside Multiservice Data Manager (MDM) format. The MDM format is a character string with the format "YYYY MM DD HH MM SS". The TIMESTAMP operator can be used to convert a trap variable containing a timestamp in another format.

This attribute is not mandatory. If the translation record does not define it, the SNMP Surveillance Adapter defines the alarm time according to the workstation time.

Fault_code

The Fault_code produces the alarm fault code which is also known as the NTP index. This fault code is an eight hexadecimal character code assigned to the type of problem reported by the alarm. Usually the first four characters identify a fault code range assigned to a specific device type, and the last four characters identify the problem using a device-specific encoding.

This attribute is mandatory in each translation record. If it is not defined by the translation record, the trap is discarded.

The following conventions are recommended for alarms defined by Nortel Networks:

- use “C” as the first character; this is the range reserved for the SNMP Surveillance Adapter
- use the value of the selected device type as the next three digits
- use the last four digits to identify the problem

Example

```
Fault_code: E "C1310001"
```

This rule uses the E-expression to create a fault code. C represents the SNMP Surveillance Adapter; 131 is the device type value for a BPS 2000 device; 0001 is the code assigned to the RMON alarm.

Customer-defined alarms

The SNMP Surveillance Adapter has a CD (customer-defined) labelling feature. If you create an alarm code with only six digits, CD is automatically added to the start of the six-digit code. This enables the code to be easily recognized in any alarm list. For example, an alarm code of 123456 becomes CD123456.

The following conventions are recommended for customer-defined alarms:

- use six digits so that CD will be automatically added to the start of the code
- use the first two digits of the six-digit code to define the selected device type; these should match the last two digits of the device type, for example, use 05 for devType 905

- use the last four digits to identify the problem

Alarm Help allows you to create a fault code and a description for an alarm. CDXXXXXX is already supported. To create an alarm fault code and description, see the 241-6001-804 *Preside MDM Workstation Utilities User Guide*.

Event

The Event rule specifies which one of the following types of information the alarm reports:

- the occurrence of a problem (SET)
- the resolution of a problem reported previously (CLEAR)
- some other information for the network operator (MESSAGE)

The translation record should define this attribute. If it is omitted, MESSAGE is used and the alarm generated will not have the required impact on the component state.

Example

```
Event: E "set"
```

This rule uses the E-expression to identify the alarm as a SET alarm.

Severity

The Severity rule specifies the alarm severity. The value of this rule must be one following:

- critical
- major
- minor
- warning
- cleared
  The “cleared” value is mandatory for, and can only be used in, CLEAR alarms.

- indeterminate
  The “indeterminate” value can only be used in MESSAGE alarms. Note that MESSAGE alarms can have any value except “cleared”.

The translation record should define this attribute. If it is omitted, “indeterminate” is used. If the alarm is a SET alarm, an “indeterminate” value can cause the alarm to be subsequently rejected.

When a SET alarm is generated for an SNMP device component, the alarm is put on the active alarm list (AAL) which is mapped to a corresponding Preside Multiservice Data Manager (MDM) component state. The following table shows the mapping between the AAL contents and MDM raw component states.

Table 14
Comparison of AAL contents and MDM component states

Active alarm list contents	Raw component states
no alarms	in-service (INSV)
only warning and/or minor alarms	in-service troubled (ISTB)
any major or critical alarms	out-of-service (OOS)

Example

```
Severity: E "major"
```

This rule uses the E-expression to identify the severity of a SET alarm as “major”.

Comment

The Comment rule produces an attribute which contains textual information intended for the network operator. The comment is usually a brief description of the event reported.

If this rule is omitted, the alarm has no comment attribute.

Example

```
Comment: E "Link down trap"
```

This rule uses the E-expression to explain that a link down trap caused the alarm.

AlmType

The AlmType rule produces an attribute which classifies the reported event in one of the following OSI-defined categories:

- communications
- qualityOfService
- processing
- equipment
- environment
- security
- operator
- debug
- unknown

If this rule is omitted, “unknown” is assigned to the alarm attribute.

Example

```
AlmType: E "communications"
```

This rule uses the E-expression to classify the alarm as a communications problem.

ProbCause

The ProbCause rule produces an attribute which classifies the reported problem. The possible values are a subset of an OSI-defined enumeration, as listed in the following table.

If this rule is omitted, the category probCauseUnknown is assigned to the alarm attribute.

Values for ProbCause from OSI-defined enumeration		
lossOfSignal	lossOfFrame	framingError
localTransmissionError	remoteTransmissionError	callEstablishmentError
degradedSignal	commSubsystemFailure	commProtocolError
lanError	dteDceInterfaceError	responseTimeExcessive
queueSizeExceeded	bandwidthReduced	retransmissionRateReduced
thresholdCrossed	performanceDegraded	congestion
atOrNearCapacity	storageCapacityProblem	versionMismatch
corruptData	cpuCyclesLimitExceeded	softwareError
softwareProgramError	softwareProgramTermination	fileError
outOfMemory	underlyingResourceUnavailable	applicationSubsystemFailure
configurationError	powerProblem	timingProblem
processorProblem	datasetModemError	multiplexorProblem
receiverFailure	transmitterFailure	outputDeviceError
inputDeviceError	ioDeviceError	equipmentFailure
adapterError	duplicateInfo	infoMissing
infoModification	infoOutOfSequence	unexpectedInfo
denialOfService	outOfService	proceduralError
otherOperational	cableTamper	intrusionDetection
otherPhysical	authenticationFailure	breachOfConfidence
nonRepudiationFailure	unauthorizedAccess	otherSecurityService
delayedInfo	keyExpired	outOfHoursActivity
operationalCondition	debugging	probCauseUnknown
inactiveVirtualCircuit	networkServerIntervention	

Example

```
ProbCause: E "lossOfSignal"
```

This example is from a link down trap. The rule uses the E-expression to classify a link down trap as a loss of signal.

ClrScope

Normally, the scope of a CLEAR alarm is restricted to the component identified by the Comp_id attribute. The SET alarm that is cleared has the same fault code attribute as this CLEAR alarm. However, the ClrScope rule provides for exceptions.

The ClrScope rule produces an optional attribute that should only be used for CLEAR alarms. If present, the ClrScope attribute indicates that the alarm is intended to clear more than a SET alarm with the same fault code currently active on the same component. The following values are valid:

- clearBase: This clears all the SET alarms currently active against this component.
- clearHier: This clears all the SET alarms currently active against this component, its subcomponents and all its related components.
- clearNCSBase: This clears the SET alarms currently active against this component if their fault codes match the NCS pattern of this alarm fault code.

  An FF pair of digits on an odd position boundary matches any pair of digits; any other digits must match exactly. The FF pair must begin in the first, third, fifth, or seventh position. For example, “1234FFFF” clears all alarms with a fault code that begins with “1234”. The pattern “5678FF12” clears all alarms with a fault code that begins with "5678" and ends with “12”.

- clearNCSHier: This clears the SET alarms currently active against this component, its subcomponents, and all its related components if their fault codes match the NCS pattern of this alarm fault code.

Example

```
ClrScope: S clearHier
```

This example is from a cold start trap. The rule uses the S-expression to clear all alarms against the component identified by the Comp_id attribute and all its subcomponents.

RelComp

Note: This rule only applies to Passport. Consider implications carefully before using it for other devices.

The RelComp rule produces an optional attribute which contains the component identifier of a related component. This requires that any hierarchical clear applied to the identified related component also be applied to this component.

If this rule is omitted, the alarm has no related component attribute.

CustId

Note: This attribute has historical value only and can be ignored. It is related to an old VPN convention.

The CustId rule produces an optional attribute assigning a customer-identifier numerical tag to the component associated with the alarm.

If the rule is omitted, an attribute with a value of 0 is created.

NotifId

The NotifId rule produces an attribute that is a number identifying a specific instance of a trap/alarm among a group of similar ones. This value should uniquely identify an alarm on a per-device basis.

If the rule is omitted, the SNMP Surveillance Adapter generates a value. If the trap contains a usable variable, a supplied attribute is recommended because it improves handling of redundant alarm feeds. However, traps do not often contain such variables.

AlmAgeMin

The AlmAgeMin rule allows old outstanding alarms to be cleared. It causes an alarm to be discarded if it does not reoccur after a specified period of time.

If the same alarm is issued against the same component before it has reached this aging time limit, the waiting period is reset. The alarm is cleared only after it has been active for the specified period from the last repetition.

The SNMP Surveillance Adapter adds the AlmAgeMin attribute to the alarm obtained from the trap translation. When an alarm with the AlmAgeMin attribute is stored in the SNMP Management Data Router (SMDR), SMDR creates a timer for the alarm. When the alarm has aged, the timer triggers SMDR to clear the alarm and SMDR issues the corresponding CLEAR alarm.

The AlmAgeMin attribute setting is in minutes. The minimum value allowed for the attribute is four minutes.

> ***Note:*** Use aging attributes only for alarms issued against dynamic components, which are components that are not polled. Active alarms for polled components are cleared by the first polling operation that discovers the component state is back to normal. This discovery usually occurs long before the alarm has aged.

Example

```
AlmAgeMin: E 6
```

The rule uses the E-expression to set the alarm aging time to six minutes. This causes the CLEAR alarm to be issued six minutes after the SET alarm is issued. If the trap is resent after three minutes, the original time period restarts and the trap is discarded six minutes after the resend, or nine minutes after the original issue.

AlmAge

The AlmAge rule allows old outstanding alarms to be cleared. It causes an alarm to be discarded if it does not reoccur after a specified period of time.

If the same alarm is issued against the same component before it has reached this aging time limit, the waiting period is reset. The alarm is cleared only after it has been active for the specified period from the last repetition.

The SNMP Surveillance Adapter adds the AlmAge attribute to the alarm obtained from the trap translation. When an alarm with the AlmAge attribute is stored in the SNMP Management Data Router (SMDR), SMDR creates a timer for the alarm. When the alarm has aged, the timer triggers SMDR to clear the alarm and SMDR issues the corresponding CLEAR alarm.

The AlmAge attribute setting is in hours. The minimum value allowed for the attribute is one hour.

> ***Note:*** Use aging attributes only for alarms issued against dynamic components, which are components that are not polled. Active alarms for polled components are cleared by the first polling operation that discovers the component state is back to normal. This discovery usually occurs long before the alarm has aged.

Example

```
AlmAge: E 6
```

The rule uses the E-expression to set the alarm aging time to six hours. This causes the CLEAR alarm to be issued six hours after the SET alarm is issued. If the trap is resent after three hours, the original time period restarts and the trap is discarded six hours after the resend, or nine hours after the original issue.

Flow control

The following rule labels perform flow control of the translation record:

- "Next" (page 213)
- "Quit" (page 214)

Next

The Next rule is used to jump to the next trap translation record in the configuration file. This rule causes the SNMP Surveillance Adapter to stop executing the rules in the current trap translation record and start executing the first rule in the next record associated with the same (Trap_oid, Trap_code) pair. The rule syntax is:

```
Next: E <E-expression>
```

where:

`E-expression` produces a Boolean value specifying if the jump to the next record occurs. If there is no next record for the same (Trap_oid, Trap_code) pair, this rule has the same effect as reaching the end of a translation record; an alarm is created based on the attributes already defined.

If the jump to the next record occurs, this record is treated as an extension of the current record. The next record

- has access to the same trap variables
- can use a name associated with a trap variable by the current record
- can use a local variable accessible in the current record
- does not have to recompute alarm attributes already computed by the current record

Quit

The Quit rule performs the same function as the Quit command used for response handling; see “Quit command” (page 100). These Quit functions conditionally stop the trap translation process and no alarm is generated. The rule syntax is:

```
Quit: E <E-expression>
```

Variable definition

The following rule labels define variables in the translation record:

- “Name” (page 214)
- “Assign” (page 214)

Name

The Name rule associates a variable name with a trap variable. This association enables the trap variable to be used in E-expressions in the same way that polled variables are used in response-handling expressions. The rule syntax is:

```
Name: <variable name> {<variable OID> | <variable
       index>}
```

Example

```
Name: port 3
```

This rule associates the port variable name with the fourth trap variable (index = 3). The alarm component identifier can be subsequently produced by:

```
Comp_id: E SPCAT(name, "PO", port)
```

For more information, see “Comp_id” (page 204).

Assign

The Assign rule performs the same function and syntax as the Assign command used for response handling; see “Assign command” (page 107). These Assign functions associate a string value with a variable that has a scope which is the translation of the current trap. The rule syntax is:

```
Assign: <variable name> <expression>
```

Special actions

The following rule labels perform special actions in the translation record:

Discover

The Discover rule is similar the Discover command used for response handling; see "Discover command" (page 104). Both the rule and the command conditionally trigger a discovery polling cycle. However, the rule does not stop trap translation while the command does stop processing of the current response. The rule syntax is:

```
Discover: E <E-expression>
```

Example

```
Discover: E TRUE
```

This example is from a cold start trap. The rule uses the E-expression to trigger device rediscovery when a cold start trap occurs.

Poll

The Poll rule is similar to the Discover rule; both rules conditionally trigger polling cycles. The Poll rule conditionally triggers a state polling cycle. The rule syntax is:

```
Poll: E <E-expression> [<delay time> [<group number>
[<component type> [<distance number>]]]]
```

where:

`nodeType` is the category name for the device.

`devName` is the device name.

`delay time` is an integer expression that specifies the delay, in seconds, before the request is scheduled. If the value of this parameter is 0, the request is scheduled with the current time and is added to the ready request queue. If the value of this parameter is greater than 0, the current time is incremented

by the specified value and the request is added to the ready request queue. If this parameter is not specified, none of the following parameters can be specified.

`group number` is the identifier of the polling request group that must be scheduled on demand. The only group number that cannot be specified is the trap polling group.If this parameter is not specified, neither of the following parameters can be specified.

`component type` is an integer expression that identifies the request that must be scheduled on demand. When you specify this parameter, only the request group with the matching attribute is scheduled. If this parameter is not specified, the following parameter cannot be specified.

`instance number` is a string expression that specifies the instance to be added to each polled variable table OID to poll a single row. This value must be entered in OID format unless the table has only one integer in the index. If this parameter is not specified, the entire table is polled.

Example

```
Poll: E TRUE
```

This example is from a reset event trap. The rule uses the E-expression to trigger a state polling cycle when this trap occurs to establish the impact of the event reported.

Cache

The Cache rule is similar to the Cache command used for response handling; see “Cache command” (page 116). Both the rule and the command create or update a device cache entry. However, the optional keep flag parameter defaults to “keep” for the Cache rule and “clean” for the Cache command. The rule syntax is:

```
Cache: <entry name> <entry value> [<keep flag>]
```

Example

```
Cache: SPCAT(cid, fc) notif
```

This rule can be used to cache the Notification ID assigned by the configuration record to a SET alarm with a fault code of “fc” raised against the component identified by “cid”. In this example, a subsequent trap

signaling the resolution of the problem using only the Notification ID contained in the original trap can retrieve this cache entry and generate the correct CLEAR alarm.

Accumulate

The Accumulate rule counts traps within a specified time interval. The repetition count may decrease when a trap is received out of the specified interval. The repetition count may also increase when a new trap is received within the interval. When the repetition count reaches the specified threshold, an internal trap is sent. The internal trap is processed immediately after the current trap is processed. The translation rule specifies a code used to define the trap specific code (V1) or the trap notification objectID (OID) (V2). The trap variable list contains the component ID, repetition count, interval and threshold1.

> ***Note:*** Use the Accumulate rule after the Comp_id rule. Otherwise, the Accumulate rule does not know which component the incremental trap is against.

The rule syntax is:

```
Accumulate: <trap code> <interval> <threshold1>
[Incremental <threshold2>] [setClear
<threshold3>][retain]
```

where:

`trap code` is the specified code for SNMPv1 or the last OID element for SNMPv2 of the internal trap sent when threshold1 is reached. The trap code value cannot be zero.

`interval` is the time interval monitored in seconds. The interval value cannot be zero.

`threshold1` is the number of trap repetitions required within the specified interval to cause an internal trap to be sent. The threshold1 value cannot be zero.

`Incremental <threshold2>` is when threshold1 is reached, and the internal trap is sent. The repetition count also keeps its current value, and threshold2 is added to the previous threshold to define the next new threshold. Each time the new threshold is reached, an internal trap is sent. The value for threshold2 must be supplied if incremental is present. The value for threshold2 is the number of trap repetitions that are added to the previous threshold to define the next new threshold. “Incremental” is not case sensitive. The threshold2 value cannot be zero.

`setClear <threshold3>` is when a separate clear threshold is defined and an internal clear trap is sent. The internal trap will be sent after threshold1 is reached at least once, and enough traps become older than the specified interval to reduce the count to clear threshold3. If the setClear parameter is supplied, threshold3 must be present. “setClear” is not case sensitive. The value for threshold3 must be less than threshold1.

`retain` is when the connection between GenDCD and the managed devices is lost or when the device is down. The accumulate trap’s count is not set to zero. The incremental count does not change when the retain parameter is not presented, and GenDCD clears this incremental count. You can use this optional parameter in any order with the Incremental and setClear optional parameters.

Optional parameters

In the Accumulate rule, the parameters [Incremental <threshold2>] and [setClear <threshold3>] are optional. The different combinations of using these parameters consist of four modes. These modes specify the behaviour of the trap accumulation after threshold1 is reached.

- `<once>` is the default mode when <Incremental> and <setClear> are not specified. When threshold1 is reached, the internal trap is sent, and the repetition count returns to zero. The next trap restarts the counting from one. When the repetition count reaches zero, the accumulation object is destroyed, and no internal trap is sent. See “Once mode example” (page 219) for more information.
- `<Incremental>` triggers the internal trap each time the new incremental threshold is reached. The repetition count keeps its current value and continues to grow if the accumulation trap is received within the interval.

When the count goes below its previous set threshold before it has taken effect, an internal trap is triggered. See “Incremental mode example” (page 219) for more information.

- <setClear> is triggered once the repetition count reaches threshold1 at least once, and decreases to reach threshold3. After threshold3 is reached, the repetition count stays at its current value and increases if the accumulation trap is received. After the count reaches the set threshold again, the clear threshold is becomes valid. When the clear threshold is reached, an internal trap is sent. See “setClear mode example” (page 220) for more information.
- <Incremental> and <setClear> combination. When the <Incremental> parameter is used with the <setClear> parameter, the functionality of both modes are combined. The threshold2 defines the next incremental threshold and threshold3 defines the clear threshold. Using these two parameters together is optional. See “setClear and incremental modes example” (page 220) for more information.

Once mode example

```
Accumulate: 32 60 12
```

This rule triggers an internal trap with trap code 32 if one component receives 12 traps within 60 seconds. When GenDCD executes this rule, it creates an accumulation object if it does not exist. If the accumulation object does exist, it increases the repetition count for the existing accumulation object. if the repetition count of this object reaches threshold 12, an internal trap is created and this accumulation object is destroyed.

Incremental mode example

```
Accumulate: 32 60 12 Incremental 4
```

When you execute this rule, the accumulation object is created and the repetition count is recorded. This rule first triggers an internal trap with trap code 32 if one component receives 12 traps within 60 seconds, but the accumulation object is not destroyed. Each time a new threshold is reached within 60 seconds, an internal trap is sent with the same trap code 32. the new incremental threshold is defined by the current threshold plus four. For example, when first threshold 12 is reached, the next new threshold is 16 (12 + 4). This means that when one component receives 16 traps within 60

seconds, it sends another internal trap with trap code 32. Then a new threshold 20 (16 + 4) is defined. A new trap is sent each time this new condition is met. If the repetition count falls down, nothing happens.

setClear mode example

```
Accumulate: 32 60 12 setClear 4
```

When you execute this rule, it creates the accumulation object. This rule first triggers an internal trap with trap code 32 when one component receives 12 traps within 60 seconds, but the accumulation object is not destroyed. Each time the repetition count falls down and reaches four within 60 seconds, an internal trap is sent with trap code 32. A clear trap is triggered when the repetition count falls down to four within 60 seconds. If the repetition count continues going up within 60 seconds, nothing happens.

Incremental and setClear modes example

```
Accumulate: 32 60 12 Incremental 4 setClear 2
```

When you execute this rule, it creates the accumulation object to track the repetition count. This rule triggers one trap with trap code 32 when one component receives 12 traps within 60 seconds. The accumulation object stays, and a new set threshold is defined by adding the current threshold to Incremental parameter 4. For example, a new set threshold is defined by 16 (12 + 4), and a clear threshold is set by setClear parameter 2. When the new set threshold 16 is reached within 60 seconds, a new internal trap is sent with trap code 32. Each time a new set threshold (for example, 20, 24, 28) is reached within 60 seconds, a new internal set trap is sent. When the repetition count decreases within 60 seconds, a clear internal trap is sent if the repetition count reaches setClear threshold2.

setClear and incremental modes example

```
Accumulate: 32 60 12 setClear 2 Incremental 4
```

This example is the same as “Incremental and setClear modes example” (page 220). The order of the setClear and Incremental parameters makes no difference. GenDCD ignores the order of the parameters, and treats them as the same command because they have the same Accumulation parameters.

retain example

```
Accumulate 32 60 3 Incremental 2 setClear 1 retain
```

In this example, the count stays if the device is lost.

```
Accumulate 32 60 3 Incremental 2 setClear 1
```

In this example, the count is set to the default value of zero.

```
Accumulate 32 60 3 retain Incremental 2 setClear 1
```

This example is the same as the first example. It does not matter where the retain parameter is located.

SNMPv1 internal trap translation record example

```
Trap_oid: ALL
Trap_code:  -5
Comp_id: E DEVNAME()
Fault_code: S C0000004
Severity: S warning
Event: S message
AlmType: S security
ProbCause: S authenticationFailure
Comment: S Authentication failure trap
Accumulate: 32 60 12 Incremental 4 setClear 2

Trap_oid: 1.2.3.4.5.*
Trap_code: 32
Comp_id: E VAR(0)
Fault_code: S C0000099
Assign: count VAR(1)
Severity: E [ EQ(count, 2)    ? "cleared"   \
              EQ(count, 12)   ? "warning"   \
              EQ(count, 16)   ? "minor"     \
              EQ(count, 20)   ? "major"     \
              GT(count, 20)   ? "critical"  ]
Event: E [ LE(count, 2)       ? "clear"     \
           GT(count,2)        ?  "set" ]
AlmType: S communications
ProbCause: S authenticationFailure
Comment: S This is an internal trap translation record
example.
```

In the first block, the -5 Trap_code identifies an authenticationFailure. The Accumulate rule indicates the triggering of an internal trap with Trap_code 32 when GenDCD receives an authenticationFailure trap 12 times within 60 seconds. Each time the new set threshold (for example, 16, 20, 24, or 28) is reached, a new internal trap is sent with Trap_code 32. When the repetition count increases, the alarm severity changes from warning to critical. When the repetition count decreases to two within 60 seconds, a clear internal trap is sent with Trap_code 32. The internal trap translation record clears the alarm that was previously sent.

The second block contains the internal trap translation record, which is sent with Trap_code 32. The alarm's severity is dependent on the second variable for the trap variablebindings. The internal trap contains the variablebindings of compid, count, interval and threshold. Therefore, the second variable VAR(1) returns the repetition count value with the higher count, and the alarm's severity changes from warning to minor, and then to critical. When the device has too many authenticationFailure traps, the device is set to a different state according to the repetition count of the authentication traps. The Event attribute may be clear or set based upon the count variable. When the setClear threshold is reached after the set internal trap is sent, the internal trap translation record clears the previous alarm. This occurs because the trap translation block takes the clear value for the Event attribute when the repetition count is less than or equal to two.The Comp_id attribute comes from the first variable of the internal trap variablebindings. You can usually use the same compid as the variable in the internal trap by reading the first variable in the trap variablebindings using VAR(0), as shown in the example. An internal trap is triggered against the same component when the component receives enough incremental traps.

SNMPv2 internal trap translation record example

```
Trap_oid: 1.3.6.1.6.3.1.1.5.5
Trap_code:  v2
Comp_id: E DEVNAME()
Fault_code: S C0000004
Severity: S warning
Event: S message
AlmType: S security
ProbCause: S authenticationFailure
Comment: S Authentication failure trap
Accumulate: 32 60 12 Incremental 4 setClear 2
```

```
Trap_oid: 1.2.3.4.5.0.32
Trap_code: v2
Comp_id: E VAR(0)
Fault_code: S C0000099
Assign: count VAR(1)
Severity: E [ EQ(count, 2)    ? "cleared"   \
              EQ(count, 12)   ? "warning"   \
              EQ(count, 16)   ? "minor"     \
              EQ(count, 20)   ? "major"     \
              GT(count, 20)   ? "critical"  ]
Event: E [ LE(count, 2)       ? "clear"     \
           GT(count,2)        ?  "set" ]
AlmType: S communications
ProbCause: S authenticationFailure
Comment: S This is an internal trap translation record
example.
```

In this example, the last element of the Trap_oid is used in the Accumulate rule. The internal trap record is identified by <Enterprise_OID>.0.<trap_code>. VAR(0) skips the two standard variables and begins from the compid variable. If you want to read starting from the SysUpTime variable, use the VAR2() function.

Expressions used in translation records

The SNMP Surveillance Adapter framework supports the following expression formats:

- “E-expression” (page 224)
- “S-expression” (page 224)
- “V-expression” (page 225)
- “C-expression” (page 225)
- “P-expression” (page 226)

See also “Expression examples” (page 227).

Note: Although V-expressions and C-expressions are supported for backwards compatibility, E-expressions can be used for all of the same functions. The E-expression format is the simpler and recommended format. The SNMP Surveillance Adapter cannot use P-expressions because they require device-specific code.

E-expression

The power of the expressions defined for polling and response handling is made available for trap translation through E-expressions. The syntax is:

```
E <expression>
```

where:

`expression` is an expression using the syntax and the operators defined for response handling.

Example

```
Fault_code: E [EQ(GENERIC(),0) ? "C0000000" \
               "C0000001"]
```

This example is from a translation record for warm and cold start traps. The E-expression is used to select the alarm fault code based on the trap generic code.

CAUTION

Cannot always access cached information

E-expressions can access cache entries where information discovered by the last discovery polling cycle is stored. However, a discovery polling cycle cannot always be performed before any given trap is received. Therefore, trap translation records that depend on cached information should be written to handle the possible unavailability of the information.

S-expression

The S-expression format directly specifies the character string that is the expression value. The syntax is:

```
S <character string>
```

This S-expression is equivalent to the E-expression:

```
E "<character string>"
```

Example

```
S message
```

This S-expression can be used to produce a value for the event alarm attribute. MESSAGE is a valid event state for an alarm.

V-expression

Note: Although V-expressions are available, E-expressions can be used for the same functions. The E-expression format is the simpler and recommended format.

The V-expression format specifies that the expression value is the value of a trap variable converted to a character string. The syntax is:

```
V {I|S|O} <variable objectId>
```

This V-expression is equivalent to the E-expression:

```
E VAR(<variable objectId>)
```

where:

`I|S|O` indicates the variable type. Use one of these three letters. `I` represents the integer, `S` represents the string, and `O` represents the object identifier.

Example

```
Comment: V S 1.3.6.4.1.562.12.1.3.1.6.0
```

This V-expression is equivalent to the E-expression:

```
Comment: E VAR(1.3.6.4.1.562.12.1.3.1.6.0)
```

C-expression

Note: Although C-expressions are available, conditional E-expressions can be used for the same functions. The E-expression format is the simpler and recommended format.

The C-expression format specifies that the expression value is the value produced by selecting one of several possible results based on the value of a trap variable. The syntax is:

```
C <variable objectId> <select>=<value>
[;<select>=<value>]*
```

This C-expression is equivalent to the E-expression:

```
E [EQ (VAR(<variable objectId>) <select1>) ? <value1> \
   EQ (VAR(<variable objectId> <select2>) ? <value2> \
...
```

This expression can be simplified and optimized by assigning the value of the "VAR (<variable objectId>)" to a local variable first.

where:

`variable objectId` is the object identifier of the trap variable used as a selector. This variable must be an integer variable.

`select` is one of the possible values for the trap variable, or "D" for a default selection.

`value` is the corresponding string value to be returned by the expression.

Example

```
Event: C 1.3.6.4.1.562.12.1.3.1.7.0 1=set; 2=clear; \
        D=message
```

This C-expression is equivalent to the E-expression:

```
Assign: event VAR(1.3.6.4.1.562.12.1.3.1.7.0)

Event: E [EQ (event,1) ? "set" \
          EQ (event,2) ? "clear" \
          "message"]
```

P-expression

The P-expression format can only be used by device-specific data collection daemons (DCDs). This format specifies that the expression value is returned by a device-specific procedure. This expression is based on the rule label and gains access to the trap contents. The syntax is:

```
P
```

Example

```
Comp_id: P
```

This P-expression can be used to call a device-specific procedure to derive the component identifier attribute from the trap contents. The SNMP Surveillance Adapter cannot do this because it does not have any device-specific code.

Expression examples

Note: Some examples in this chapter contain numbers at the start of each line. These numbers are not part of the actual request; they have been added for clarity purposes to refer to individual lines.

Translation record example 1:

```
1   Trap_oid: 1.3.6.1.2.1
2   Trap_code: -1, -2
3   Comp_id: E DEVNAME()
4   Fault_code: E [EQ(GENERIC(),0) ? "C0000000" \
                   "C0000001"]
5   Severity: S cleared
6   Event: S clear
7   AlmType: S operator
8   ProbCause: S operationalConditions
9   Comment: E [EQ(GENERIC(),0) ? "Cold start trap" \
           "Warm start trap"]
10   ClrScope: S clearHier
11   Discover: E TRUE
```

Translation record example 1 is from a repeater trap configuration file, and it defines the translation rules for cold and warm start traps. Negative trap codes identify these traps as generic by the rule (code=-(generic+1)). The expressions in this record are used to

- define the alarm component identifier as the name currently associated with the device (line 3)
- select the alarm fault code based on the trap generic code (line 4). If the generic code is 0 (cold start trap), the fault code is C0000000; otherwise the fault code is C0000001.
- set the alarm severity to “cleared” (line 5)
- set the alarm event type to CLEAR (line 6)

- define the alarm type as operator (line 7)
- define the alarm probable cause as operationalConditions (line 8)
- select the alarm comment based on the trap generic code (line 9)
- specify that this alarm clears all alarms currently active against the device and its components (line 10)
- specify that this alarm triggers a new discovery polling cycle (line 11)

Translation record example 2:

```
1   Trap_oid: 1.3.6.1.2.1.22
2   Trap_code: 1
3   Comp_id: E DEVNAME()
4   Fault_code: S C0510001
5   Name: state 1.3.6.1.2.1.22.1.1.2
6   Severity: E [EQ(state,2) ? "cleared" "major"]
7   Event: E [EQ(state,2) ? "clear" "set"]
8   AlmType: S communications
9   ProbCause: S outOfService
10   Comment: E VAR(1.3.6.1.2.1.22.1.1.3)
11   Poll: E TRUE
```

Translation record example 2 is from a repeater trap configuration file, and it defines the translation rules for a repeater health trap. The expressions in this record are used to

- define the alarm component identifier as the name currently associated with the device (line 3). The alarm is raised against the device even when it signals a port failure because the trap does not identify any subcomponent.
- select the alarm fault code based on the device type (line 4). The device type is 051, and the code assigned to this alarm is 0001.
- associate the name “state” with the trap variable reporting the new repeater state (line 5)
- set the alarm severity to “cleared” if the state variable has the value 2; otherwise a “major” SET alarm is generated (lines 6 and 7)
- define the alarm type as communications (line 8)
- define the alarm probable cause as outOfService (line 9)

- copy the alarm comment from the corresponding trap variable (line 10)
- specify that this alarm triggers a new state polling cycle (line 11). This cycle can establish the impact of the problem reported.

Functions used in translation records

The SNMP Surveillance Adapter framework uses various functions to access information in the trap being translated. For information about these functions, see the following sections in “Context access operators” (page 132):

- “VAR” (page 143)
- “VAR2” (page 144)
- “GENERIC” (page 145)
- “SPECIFIC” (page 146)
- “VERSION” (page 146)
- “TRAPOID” (page 147)
- “SYSUPTIME” (page 148)

Note 1: A trap translation record can be associated with several trap codes. In this type of record, the GENERIC and SPECIFIC functions can be used to generate code-specific values for an attribute if required.

Note 2: There is also a PROC function which is used to support P-expressions for device-specific data collection daemons (DCD). This operator is not intended for use in trap translation rules for the SNMP Surveillance Adapter.

Chapter 10
Troubleshooting the SNMP Surveillance Adapter

This section describes tools and methods for troubleshooting the SNMP Surveillance Adapter and associated components. It contains the following topics:

Debugging tools

The SNMP Surveillance Adapter is based on a data collection daemon (DCD) which has input and output processes. Debugging can come from several sources which enables tracking of input, output, and internal events. The following tools can be used for debugging purposes:

- "GMDR API" (page 236)
- "GMDR Administration tool" (page 236)

-v command line option

See "Using the DCD in verification mode" (page 34).

Trap sniffer

The trap sniffer is a diagnostic tool used to record all traps emitted with specified object identifiers (OIDs) and/or from specified addresses. The trap sniffer can indicate if there are any traps being emitted by some devices that are not being handled correctly by their corresponding data collection daemon (DCD) processes.

Startup command

Use an Preside Multiservice Data Manager workstation to start the trap sniffer process. You must be logged in as root to run the trap sniffer process. Before you run the trap sniffer process, enter the following command:

```
unmask 077
```

The startup command for the trap sniffer process is:

```
/opt/MagellanNMS/bin/tsnf <command line options>
```

where command line options are:

`-a <ip address filter>` This parameter specifies ip addresses where all traps emitted are recorded. The default is "*", which represents all addresses. The syntax is the same as the syntax of the addrFilter DCD option; see addrFilter in "Run-time options" (page 41).

`-d <logging levels (FATAL,SNO,MAJOR,MINOR,INFO,TRACE, ALL)>` This parameter specifies the log level. More than one level can be specified. The default is `"FATAL,SNO,ERRORS,MAJOR,MINOR,INFO"`. The syntax is the same as the syntax of the -d DCD command line parameter; see the 241-6001-310 *Preside MDM Server Reference Guide*.

`-h <display help menu>` This parameter specifies that the help menu be displayed.

`-i <send logs to stdout>` This parameter specifies that logs be displayed as standard output (stdout). The syntax is the same as the syntax of the -i DCD command line parameter; see “Using the DCD in verification mode” (page 34).

`-l <log file location>` This parameter specifies the log file location. The default is /opt/MagellanNMS/data/tsnf.log.

`-o <enterprise oid filter>` This parameter specifies enterprise OIDs for the trap sniffer to record. The default is "*", which represents all traps. The syntax is:

`-o <enterprise OID> -n <profile name>`

`-o <enterprise OID>` specifies the enterprise OID of the device type. For example, iBWA 5100 base stations are `-o 1.3.6.1.4.1.562.21.1.1`.

Trap log formats

The log output format for SNMPv1 traps is different from the format for SNMPv2 traps, as shown in the log output tables. Note the following comparisons of the two tables:

- Each line of log output starts with entries of <log level> <log date> <log time>. These are not shown in the tables for clarity purposes.
- In the first line, SNMPv1 output has the sender and agent addresses; SNMPv2 output has the IP address and notification OID instead.
- In the second line, SNMPv1 output has the enterprise OID and the generic and specific codes; SNMPv2 does not have this output.
- The format of the next two lines are the same for SNMPv1 and SNMPv2 traps.

SNMPv1 log output
`PDU: Sender Addr=<ip address> Agent Addr=<ip address>`
`PDU: Oid=<enterprise oid> Gen=<generic code> Spec=<specific code>`
`PDU: Time=<trap timestamp> Community=<community string>`
(Sheet 1 of 2)

SNMPv1 log output
`VAR: Oid=<variable oid> Type=<variable type> Val=<value>` ...
(Sheet 2 of 2)

SNMPv2 log output

`PDU: Addr=<ip address> Notification OID=<notification oid>`
`PDU: Time=<trap timestamp> Community=<community string>`
`VAR: Oid=<variable oid> Type=<variable type> Val=<value>`
...

Trap server trace

The trap server (TSVR) trace utility is a diagnostic tool for the trap server process. This utility can be used to extract useful contents of a trap and send these contents to a log file so that a problem can be isolated quickly. It enables you to capture trap contents before trap server processing, and provides a trace of events that occurred within the trap server.

Starting the trap server trace utility

Use this procedure to turn the trap server trace utility on and off. It is not normally necessary to stop and restart the trap server.

1. If the trap server is not already running, use one of the following methods to start it:
   - Start the Preside Multiservice Data Manager Server Administration tool. Select the trap server to start.
   - Open a Unix window and use the same window for the entire procedure. From the command line within the Unix window, type

     `tsvr`
2. To determine the trap server process identification number, type

   `ps -ef | grep tsvr`
3. To turn on the trap server trace utility, type

kill -USR1 <tsvr process id>

Press the Return key. The following response appears in the log file:

```
trap trace flag is on
```

4 When the trap server receives a trap, it will output some useful contents of the trap to the trap server log file. To view these trap contents, see the log file named tsvr.log.

5 To turn off the trap server trace utility, type

kill -USR1 <tsvr process id>

Press the Return key. The following response appears:

```
trap trace flag is off
```

Note: This same kill command is used to turn the trace utility on and off. If the utility was off, the command will turn it on. If the utility was on, the command will turn it off. To verify the state of the utility, see the response in the log file.

DCD API

The DCD application programming interface (API) can be used to

- discover the database contents of components
- monitor DCD output including alarms, state change notifications, and server notifications

For details, see “DCD API requests” (page 236).

SMDR API

The SNMP Management Data Router (SMDR) API can be used to

- discover the database contents of components
- monitor SMDR output including alarms, state change notifications, and server notifications

The same commands are supported for DCD API and SMDR API. The startup command for SMDR API is

/opt/MagellanNMS/bin/smdrapi

GMDR API

The General Management Data Router (GMDR) API can be used to

- identify components that are currently known
- list active alarms
- trace data produced by GMDR
- determine which SMDR servers are connected

The startup command for GMDR API is

```
/opt/MagellanNMS/bin/gmdrapi
```

GMDR Administration tool

The GMDR Administration tool is a graphical user interface (GUI) that is easier to use than the GMDR API. The information provided by the GUI is less detailed but it may suffice. This GUI can be used to

- list server processes and their connection states
- list known components and their states
- provide a statistics summary
- determine which SMDR servers are connected

The GMDR Administration tool is normally started from the Preside Multiservice Data Manager toolset. The startup command is

```
/opt/MagellanNMS/bin/gmdradmin
```

DCD API requests

The data collection daemon (DCD) Application Programming Interface (API) is similar to the Alarm and Status API; see the 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The DCD API is mainly intended for debugging use. Injection (Inbound API) is not supported. Additional DCD-specific actions are supported. The following DCD API requests are available:

- “Register” (page 237)
- “Get” (page 237)
- “Create” (page 238)

- "Action" (page 238)

  Action requests include database reset, component delete, query property, query node name, poll, add node, delete node, unmanage node, manage node, add address, and delete address.

See also "Startup command" (page 237).

Startup command

To start the DCD API, type

```
/opt/MagellanNMS/bin/dcdapi -serv <process name>
```

where:

`process name` is the DCD executable name modified by the -n or -N parameters. For example, `gendcd -n reg1` creates a process name of `gendcd_reg1`.

Register

The register request is used by the client process, for example the API session, to register itself with the DCD. A user identification is required, and the password is ignored.

Example

```
_cmd: register
_user_id: myName
```

Get

The get request is used by the client process to retrieve information currently stored in the DCD internal database. This information can include nodes, links, and server statistics.

Example

The following is an API request for the component identifier and the raw state of the RBSE MONTREAL device and all of its subcomponents.

```
_cmd: get
_obj_class: node
_obj_id: compId NI RBSE MONTREAL
_scope: all
_attr_id: rawState
```

The following is an API request for the component identifier and the raw state of all the devices, subcomponents, and links in the DCD database.

```
_cmd: get
_obj_class: network
_obj_id: networkId S compRoot
_scope: all
_attr_id: rawState
```

Create

The create request is used by the client process to inform the DCD of its interest for some event types. This request can create a sieve for alarms, state change notifications, or server reset notifications.

Examples

The following is an API request to create a sieve for alarms.

```
_cmd: create
_obj_class: sieve
_attr: eventFilter SS eventType EQ S alarm
```

The following API is a request is to create a sieve for state change notifications.

```
_cmd: create
_obj_class: sieve
_attr: eventFilter SS eventType EQ S rawStateChange
```

Action

An action request is used by a client process to ask the DCD to perform a task. DCD action requests include the following:

- “Database reset” (page 239)
- “Component delete” (page 239)
- “Query property” (page 239)
- “Query node name” (page 240)
- “triggerPoll” (page 241)
- “Add node” (page 241)
- “Delete node” (page 244)
- “Unmanage node” (page 244)

- “Manage node” (page 244)
- “Add address” (page 245)
- “Delete address” (page 246)

Database reset

The database reset (dbReset) request should be used wisely because it is associated with high process and device costs. This request is used to

- recreate the seed file
- clear the component database
- restart discovery from the seed file

Example

The following is an API request to reset the DCD component database.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: resetDB
```

Component delete

The componentDelete request is used to delete a component and all its subcomponents from the component database. The component deletion is propagated to clients of the DCD server and may be reflected in the network model. However, the component will return in the next polling cycle if it is still in the MIB.

Example

The following is an API request to delete a card and the subordinate ports in a Passport 4400 switch from the component database.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: componentDelete
_attr: compId NI MPA TORONTO CARD 4
```

Query property

The queryProperty request is used to

- search the property list of a specified component

- return the specified property value if found

This API request now supports as many properties and/or components as required per request.

Example

The following is an API request to find the IP address of a Shasta device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: queryProperty
_attr: compId NI SSG OTTAWA
_attr: propName S Ip Address
```

Note 1: Multiple propName lines can be specified to retrieve multiple properties at one time; if no propName line is specified, all available properties are returned. Similarly, multiple compId lines can be specified to retrieve properties from multiple components.

Note 2: The specified component name can represent a wild-carded module name. For example, "CES *" or "EM CENTER*" will retrieve properties from all matching modules currently reporting to the surveillance stack.

Query node name

The queryNodeName request is used to query the DCDs to obtain a device name based on the supplied IP address. The DCDs can identify all the nodes matching the supplied IP address and community string pair.

Note: A dcdQueryNodeName action script is also available; see “dcdQueryNodeName” (page 53).

Example

The following example is an API request to find the device name of an iBWA 5100 device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S GMDR
_action_type: queryNodeName
```

```
_attr: nodeAddress S 47.129.32.153
_attr: nodeCommunity S public
_attr: nodeDeviceType I 20
```

triggerPoll

The poll request is used to start a new discovery polling cycle for the specified device.

Note: A dcdTriggerPoll action script is also available; see "dcdTriggerPoll" (page 54).

Example

The following is an API request to start a new on-demand polling cycle for a Shasta device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: triggerPoll
_attr: compId NI SSG OTTAWA
_attr: delayTime I 10
_attr: groupNumber I 4
_attr: compType I 1
_attr: instNumber S 4.1
```

Add node

Note: A dcdAddNode action script is also available; see "dcdAddNode" (page 52).

The addNode request is normally used to create a new polling object (DcdAgent) based on the supplied parameters; discovery of the corresponding device is then attempted. However, if an existing polling object already has the same name and/or the same address as the values supplied, the action performed by the DCD depends on the following factors:

1 Is the device name the same as an existing DcdAgent?

2 Does the device type match an existing DcdAgent?

3 Does the device address (IP address and community string) match an existing DcdAgent?

4 Is the existing DcdAgent in a managed state?

5 Is the multiple address option enabled for the type of device specified?

6 Is the existing agent reachable?

The following table shows how these factors affect the DCD action performed.

1) Same name?	2) Device type matches?	3) Address matches?	4) DcdAgent managed?	5) Multi-address enabled?	6) Agent reachable?	DCD action performed
Yes	No					Replace DcdAgent; see Scenario A in the following section
Yes	Yes	Yes				None; see Scenario B
Yes	Yes	No	No			None; see Scenario C
Yes	Yes	No	Yes	Yes		addAddress; see Scenario D
Yes	Yes	No	Yes	No	Yes	None; see Scenario E
Yes	Yes	No	Yes	No	No	Replace DcdAgent; see Scenario F
No		Yes	No			None; see Scenario G
No		Yes	Yes			Replace DcdAgent; see Scenario H
No		No				addNode; see Scenario I

Scenarios

The following explanations correspond to the scenarios shown in the previous table.

Scenario A: This request is for an agent profile change; the old agent is deleted and a new one is created.

Scenario B: This request is a duplicate; it is ignored.

Scenario C: The device is unmanaged; this request is ignored.

Scenario D: This request is treated as an addAddress request.

Scenario E: This request is for an address change for a device that is reachable; it is rejected.

Scenario F: This request is for an address change for a device that is managed but not reachable; the old agent is deleted and a new one is created.

Scenario G: This request is for a device name change for an unmanaged device; it is rejected.

Scenario H: This request is for a device name change; the old agent is deleted and a new one is created.

Scenario I: This is a normal manual discovery request in which neither the name nor the address are currently known; a new agent is created.

Example

The following is an API request to create a DcdAgent object for a Passport 4460 device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: addNode
_attr: compId NI MPA CAL
_attr: nodeAddress S 47.130.96.35
_attr: nodeCommunity S public
_attr: nodePort I 161
_attr: nodeDeviceType I 6
```

Delete node

The deleteNode request is used to delete a device's corresponding DcdAgent object and its associated components from the DCD database, and may be reflected in the network model. However, the object and components will return if a trap is received from the device and trap-based discovery is enabled.

Example

The following is an API request to delete a DcdAgent object for a Passport 4460 device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: deleteNode
_attr: compId NI MPA CAL
```

Unmanage node

The unManageNode request is used to put the DCD agent associated with a device in an "unmanaged" state. In this state, the DCD

- stops polling the specified device
- discards traps received from the device
- puts the device and it subcomponents in an UNKNOWN state
- frees the socket used to poll the device

Example

The following is an API request for the DCD to stop managing a Passport 4400 device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: unManageNode
_attr: compId NI MPA DEV1
```

Manage node

The manageNode request is used to return a device's DCD agent to the "managed" state.

Example

The following is an API request for the DCD to rediscover a Passport 4400 device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: manageNode
_attr: compId NI MPA DEV1
```

Add address

The addAddress request is used to add a polling or trap address to the address list for the specified device. The DCD rejects this request if one of the following conditions occurs:

- The device profile does not allow this action.
- The specified address has already been assigned to another device, and this other device is currently using the address for polling.

The following attributes must be specified in the request:

- device name
- IP address
- community string
- address type (“poll” or “trap”)

Note: A dcdAddAddress action script is also available; see “dcdAddAddress” (page 55).

Example

The following is an API request for the DCD to add an address to a Passport 4460 device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: addAddress
_attr: compId NI MPA DEV4
_attr: nodeAddress S 47.20.137.84
_attr: nodeCommunity S public
_attr: nodeAddrType S poll
```

Delete address

The deleteAddress request is used to remove a polling or trap address from the address list for the specified device. The DCD rejects this request if one of the following conditions occurs:

- The device is not found.
- The specified address is not found.
- The specified address is currently being used for polling.

The following attributes must be specified in the request:

- device name
- IP address
- community string
- address type

Note: A dcdDeleteAddress action script is also available; see "dcdDeleteAddress" (page 56).

Note: You can remove all of the addresses for this device at once by using a wildcard (*) in place of the parameters for IP address, community string, and address type. If you use this option, all addresses for this device will be removed except the current active polling address.

Example

The following is an API request for the DCD to delete an address from the address list for Passport 4460 device.

```
_cmd: action
_obj_class: server
_obj_id: serverId S dcd
_action_type: deleteAddress
_attr: compId NI MPA DEV4
_attr: nodeAddress S 47.20.137.84
_attr: nodeCommunity S public
_attr: nodeAddrType S trap
```

Log files

Logs are issued by the SNMP Surveillance Adapter to signal the occurrence of error conditions and major events. Additional information can be included as an option. Logs can also be issued from response handling and trap translation records.

Messages sent to log files are grouped into different levels. You can select the appropriate level on the process command line to obtain the required message details from the following log files:

- "DCD log files" (page 247)
- "SMDR log files" (page 247)
- "TSVR log files" (page 248)

You can also cause a process to dynamically start or stop issuing logs at all levels; see "USR signals" (page 248).

DCD log files

For information about data collection daemon (DCD) exit codes and error messages, see the 241-6001-310 *Preside MDM Server Reference Guide*.

> ***Note:*** By default, the DCD errors are captured in the log file /opt/MagellanNMS/data/<process name>.log

where:
`process name` is the DCD executable name with optional modifications by the -n and -N parameters. The default process name is `gendcd`.

SMDR log files

For information about the SNMP Management Data Router (SMDR) exit codes and error messages, see the 241-6001-310 *Preside MDM Server Reference Guide*.

> ***Note:*** By default, the SMDR errors are captured in the log file /opt/MagellanNMS/data/smdr.log

TSVR log files

For information about trap server (TSVR) error messages, see the 241-6001-310 *Preside MDM Server Reference Guide*.

By default, the TSVR errors are captured in the following log files:

- /opt/MagellanNMS/data/tsvr.log
  This file contains trap server error logs.
- /opt/MagellanNMS/data/tsvrrep.log
  This file contains trap reporter (TRep) error logs.

  Note: To eliminate overhead, only FATAL, ERROR, and MAJOR messages are issued by TSVR and TRep.

USR signals

USR signals are used to record logs of particular levels as follows:

- USR1 signal is used to start recording ALL log levels.
- USR2 signal is used to go back to recording only the log levels selected at the start of the process.

Server reset notifications

The DCD sends server reset notifications reporting events that impact more than single components. These notifications often result from an action request being executed by the DCD; the server reset notification then reports this event. These notifications include

- **database reset**
  This reset is triggered by an API action request. All information is deleted from the component database, and the DCD rediscovers all devices. For information on the API request, see “Database reset” (page 239).
- **component deletion**
  This deletion is triggered by an API action request or component removal detection while polling. The component and its subcomponents are deleted from the component database and may be reflected in the network model. For information on the API request, see “Component delete” (page 239).

- **lost device**
  This notification is triggered by the device becoming unreachable. All active alarms are deleted, and the device and component polling states are set to UNKNOWN.
- **reconnected device**
  This notification is triggered by the device becoming reachable again. The device is being rediscovered.
- **device managed state**
  This notification is triggered by an action request to stop or restart monitoring a device. The polling and trap translation stops or restarts. For more information, see For information on the API requests, see “Unmanage node” (page 244) and “Manage node” (page 244).
- **discovery state done**
  This notification is triggered the first time that the polling cycle of a device is completed.

Workstation verification

See the following sections for methods used to verify workstations:

- “mnsd” (page 249)
- “IPC registration” (page 249)
- “snoop” (page 250)
- “netstat” (page 250)

mnsd

mnsd is the Preside Multiservice Data Manager server. It is required for server processes to register their service names. The command to use for verification is

```
ps -ef | grep mnsd
```

IPC registration

All processes must be “alive” and registered with IPC. The log `"COMENDPOINT CREATION Illegal name used"` indicates that the target server is not registered with IPC. The command to use for verification is

```
ipcmon -p
```

snoop

Use the UNIX utility *snoop* to verify the arrival of traps on the workstation at port 162. The command to use for verification is

```
snoop <device IP address> port 162
```

netstat

Use the *netstat* command to determine whether or not port 162 is bound by the trap server reporter (TRep). Before starting the trap server (TSVR), the following command should indicate that the port is unbound:

```
netstat -a | grep "snmp-trap"
```

If the port is bound, make sure that no other running process requires port 162. For example, is HP-Openview running?

Common error symptoms

See the following sections for most likely causes and suggested corrective actions for common error symptoms:

- “Alarms are displayed for some devices but not others” (page 250)
- “No alarms are being received” (page 251)

Alarms are displayed for some devices but not others

The most likely cause for Preside Multiservice Data Manager (MDM) to display some alarms but not others, and the order in which you should troubleshoot them are as follows:

- Physical IP connectivity has been lost between the MDM workstation running the DCD process and the device that should be sending traps. The IP link may be broken or the devices may be down.

  At the workstation running the DCD software, use the PING command to determine if connections to the devices are alive. For example: ping 37.208.155.55.

- Trap subscription is not set up on some of the devices that should be sending traps to the workstation running the DCD.
- The State command has been issued to set the state of a device to UNK (unknown). Consequently the GMDR may consider the device as unmanaged. For details, see “State command” (page 114).

No alarms are being received

The most likely causes and the order in which you should troubleshoot them are as follows:

- The device is using a community string other than “public”. For example, an EdgeLink MUX device may have its community string set to “snmp_trap”. This creates a situation in which the community string on received traps is different than the string used to poll the device.

  Make sure the device is using “public” as a community string. If it is not, perform the following steps:

  — Edit the configuration (.cfg) file in the customer location /opt/MagellanNMS/cfg/dcd

  — Set the ignoreTrapComm parameter to TRUE. For more information, see “ignoreTrapComm” (page 48).

  — Make sure the defaultComm parameter is set to “public”. For more information, see “defaultComm” (page 46).

- The trap server is not running or cannot bind to UDP port 162. Only one application can bind to port 162 at a time. Check for other applications running on the port.
- The device profile is not configured or is invalid. Verify configuration files using the -v command line option.
- IP connectivity to the MDM workstation is lost because of link failure. Use the PING command to determine if the connection to the workstation is alive.
- Trap subscription is not set up for any of the devices that should send traps to the workstation running the MDM integration software.
- The correct IP address, or host name, of the SMDR server is not registered with the GMDR Administration tool.

  For more information, see the 241-6001-303 *Preside MDM Administrator Guide*.

- Connectivity between SMDR and the DCD is not enabled.
- The GMDR server has exited.

Appendix A
Generic SNMP device support

This section explains how to integrate generic SNMP devices into Preside Multiservice Data Manager. The generic SNMP device profile provides the framework for integrating generic SNMP devices that don't require the extensive configuration and modeling required for specific SNMP devices. This section describes the generic device model, the generic SNMP device profile, the configuration files, and how to configure generic SNMP devices. See the following sections for more information:

- "Generic device model" (page 257)
- "Generic SNMP device profile" (page 254)
- "Generic DCD configuration files" (page 264)
- "Configuring generic SNMP devices" (page 271)
- "Modifying configuration files" (page 272)
- "Starting the GenDCD process" (page 273)

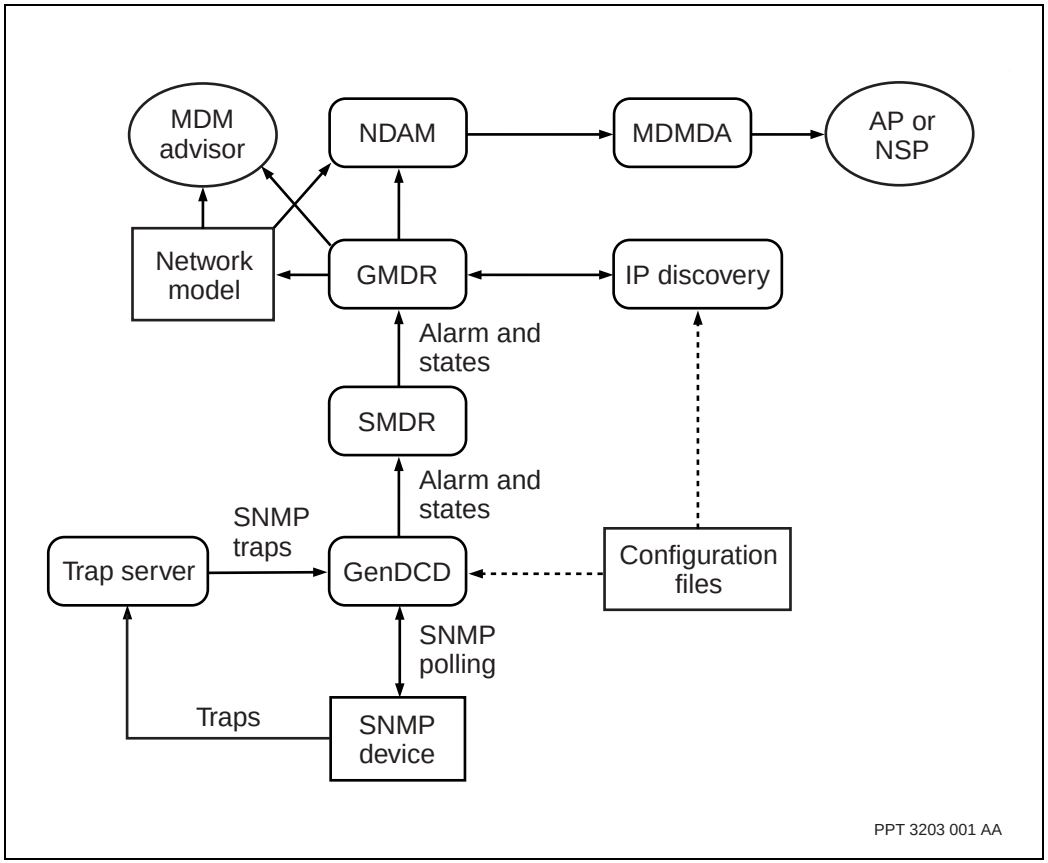
Generic SNMP device profile

Generic SNMP devices can be integrated into Preside Multiservice Data Manager (MDM) using a set of configuration files. These files define a generic SNMP device profile for the SNMP devices that are used by the Generic Data Collection Daemon (GenDCD) process.

MDM uses the data collected by the GenDCD and provides surveillance information for these generic SNMP devices through the fault toolset. This information can also be made available to AP or NSP through an MDM Device Adaptor (MDMDA). “SNMP Fault Management Architecture” (page 255) illustrates the architecture and its option to provide data to AP and NSP.

The generic SNMP device profile enables a device to be deployed in a network on a temporary or permanent basis without the detailed modeling and configuration files. All configuration files required by the generic profile are integrated into MDM fault base, and are therefore immediately available and do not require any cartridge installation (See “Generic DCD configuration files” (page 264)). The only devices excluded are those that do not support the system group and or the interface table originally defined in RFC1213.

Figure 3
SNMP Fault Management Architecture



SNMP supported

SNMPv1 and SNMPv2c are supported for both polling and trap translation.

Minimum and optional requirements

The minimal MIB requirements for the generic device SNMP agent are the system group and the interface table originally defined in RFC1213. If the device does not support these, it cannot use the generic profile.

Optionally, if the interface table extension originally defined by RFC1573 is also supported, the *ifName* column of this table extension can be used to provide more meaningful names for the device interfaces. If additional IP addresses are defined in the *ipAddEntAddr* table originally defined in RFC1213, those addresses are added to the list of IP addresses that may appear in traps sent by the device.

Traps supported

Only the ColdStart, WarmStart, LinkDown, LinkUp and AuthenticationFailure generic traps are separately translated. A single default translation handles all the other traps. See “generic.tra” (page 268) for more information on traps.

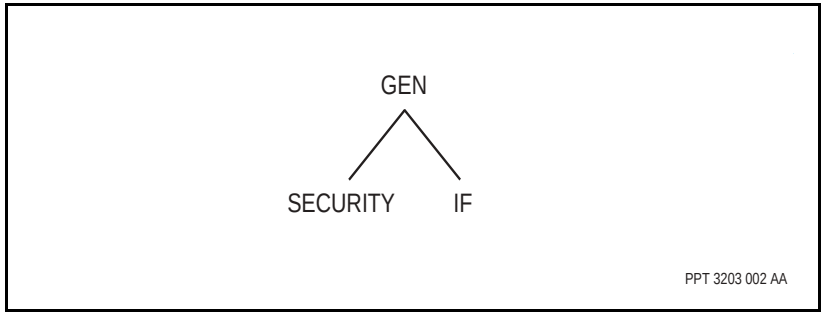
Generic device model

This section describes the generic device model and its behavior.

Device model components

The device model consists of three components as illustrated in the "Generic device model" (page 257) diagram below.

Figure 4
Generic device model



The GEN component type is associated with a component representing the device itself. Its instance is the device name, which can be obtained in several ways. See "Device name" (page 259).

The SECURITY component type is associated with an instance-less dynamic component created when an *authenticationFailure* generic trap is received.

The IF component type is associated with one of the device interfaces. By default, the instance value is the corresponding value of *ifIndex*. The customer can provide a more useful instance value in the *ifName* MIB table.

Device behavior options

The generic profile can be used for a wide range of devices with varying requirements from customers. Several options are available to provide flexibility to the generic profile.

- "Device name" (page 259)
- "Interface name" (page 259)
- "Interface raw state" (page 260)
- "Cold and warm start traps handling" (page 261)

The default value selected for each of these options produces a behavior compatible with how Preside Multiservice Data Manager (MDM) manages other devices.

The options can be enabled at either the process level or the profile level.

- If enabled at the process level, the options are specified in a customized version of the process options (.cfg) configuration file and only apply to the devices monitored using the generic profile by the corresponding GenDCD instance. In this case, copy */opt/MagellanNMS/lib/cfg/dcd/genericdcd.cfg* into the directory */opt/MagellanNMS/cfg/dcd* and modify it to enable the required options.
- If enabled at the profile level, the options are specified in a customized version of the profile option (.agp) file and apply to all GenDCD instances using this customized profile. In this case, copy */opt/MagellanNMS/lib/cfg/dcd/generic.agp* into the directory */opt/MagellanNMS/cfg/dcd* and modify it to enable the required options.
- If an option is specified in both the *.cfg* and *.agp* files, the agp specification, which is read last, applies.

  Note: These options are not designed as a substitute for MDM SNMP Surveillance Adapter capabilities. If more flexibility is required, a separate set of configuration files, possibly based on the files included in the generic profile, may need to be developed.

Device name

For many cartridges, the device name is derived from the contents of the *sysName* MIB variable. This may not be possible or desirable for some generic devices. The generic profile offers the following alternative ways to define the device name:

- the device name is derived from the sysName MIB variable; this is the default behavior if the option is not specified
- the device name is defined using the device's IP address
- the device name is obtained from the network hosts tables
- the device name is the name supplied in the Discovery command. With this option, the name will not be modified by GenDCD. (The source of this discovery command is the IP Discovery tool or the operator using the *dcdAddNode* macro or an API session).

To use one of these alternative ways to define the device name, create an entry with the key NAME_SOURCE in the process cache. The value of this entry can be SYSNAME, IPADDRESS, HOSTNAME or COMMAND to select the corresponding behavior. For example, in the .cfg or .agp file, the following line specifies that device names are defined using each device address:

```
Cache: NAME_SOURCE: IPADDRESS
```

When no explicit selection is made for this option, the device names are derived from the *sysName* variable. When the selected way to define the device name fails for a given device (for example, the *sysName* variable is left undefined in the device MIB), the device IP address is used.

Interface name

The instance associated with each IF component is the corresponding value of *ifIndex* (by default). This name is not useful to an operator or field engineer to identify the corresponding physical or logical component.

Some SNMP devices support an interface table extension originally defined by RFC1573 including an *ifName* field that can be used to provide a more useful name. The generic profile attempts to poll this table extension and looks for *ifName* values of the form "IF <better name>". When such a value is found, <better name> is used as the component instance instead of the

ifIndex value. It should contain only valid characters (letters, digits, dashes, periods, and underscores). Invalid characters are replaced by underscores and only the first 20 characters of <better name> are used.

> ***Note:*** Devices that do not support the interface table extension appear to have an empty table to GenDCD and this customization is not available.

Interface raw state

Traditionally in MDM, the raw state of a component that is not functional is *OOS*. If this state is not entered after receiving an alarm, a proxy alarm is issued to account for this state in the Alarm Display.

SNMP often identifies whether a component is non-functional due to a fault, or operator intervention. For components that are disabled by the operator, some customers do not want an alarm, although this is incompatible with normal MDM practice. However, when the target system is AP or NSP, normal MDM practice may be ignored.

For IF components, the generic profile follows the MDM practice by default; however, it can be configured to satisfy this “no alarm if administratively disabled” requirement by defining the proper process cache entry. The key for this cache entry is “OOS_FOR_ADMIN_DOWN” and the possible values are “YES” and “NO”. Therefore, if either the .cfg or agp file contain the following line, alarms are not issued when an interface is disabled by the operator:

```
Cache: OOS_FOR_ADMIN_DOWN: NO
```

If this cache entry does not exist or if its value is YES, the MDM normal practice is followed:

- the raw state of a polled interface is set to *OOS* if either *ifOperStatus* or *ifAdminStatus* does not have the value *up*; a proxy alarm is issued if no active alarm already exists
- a *linkDown trap* is immediately translated to a SET alarm against the corresponding IF component

If this cache entry exists and has the value NO:

- polling an interface produces one of the following results:

— *ifOperStatus* and *ifAdminStatus* are *up*: the raw state is set to INSV. A proxy CLEAR alarm may be issued by SMDR to clear the currently active alarms.

— *ifOperStatus* is *testing* or *ifAdminStatus* is not *up*: the raw state is set to *UNK*. All active alarms against this interface are discarded.

— all other combinations of *ifOperStatus* and *ifAdminStatus*: the raw state is set to *OOS*. A CRITICAL SET alarm is issued against the interface.

- the reception of a *linkDown* trap causes a polling request for the corresponding interface to be scheduled. No alarm is issued.

  Note: The interface problem may be corrected before the polling request is executed. In this case, the problem is not reported if the reply to the polling request shows the interface back to normal.

Cold and warm start traps handling

When devices restart, the system raises a CLEAR alarm at the device level. These alarms usually have a hierarchical scope associated with them because any problem that may have existed on the device before the restart may have been fixed while the device was unreachable. Problems existing after the restart should cause new traps or should be discovered by subsequent polling activities.

Some customers require a SET alarm when a ColdStart or WarmStart trap is received. This may be necessary because a restart may have been caused by a hidden problem. The SET alarm also increases the visibility of the restart event since CLEAR alarms are not easily visible in AP or NSP.

The generic profile uses the trap accumulation feature to detect and report repetitions of ColdStart or WarmStart traps. A SET alarm is issued within 30 minutes if the trap is repeated. The alarm severity depends on the repetition count as follows:

- 2: minor
- 3: major
- 4 and more: critical

When no additional trap has been received for the last 30 minutes, these alarms are automatically cleared. Optionally, the customer may configure the generic profile to also issue a SET alarm when an isolated ColdStart or WarmStart trap is received by defining the proper process cache entry.

The key for the cache entry is ALARM_ALL_RESTARTS and the possible values are YES and NO. Therefore, a SET alarm (WARNING severity) is issued when a non-repeated ColdStart or WarmStart trap is received if either the .cfg or the .agp file contains the following line:

```
Cache: ALARM_ALL_RESTARTS: YES
```

Duplicate IP address detection

Many cartridges have recently been modified to detect duplicate IP addresses in the network. The generic profile cannot use exactly the same technique that has been used by these cartridges since:

- in the case of a generic cartridge, there is no fixed system OID expected from all the devices
- the MIB originally defined by RFC1213 has no mandatory variable that can be used to identify the hardware instance

Therefore, the system OID is read from the MIB and stored in the device cache when the device is discovered. Subsequently, each time the node discovery or the reachability request is executed, this variable is read from the MIB and compared to the cached value; if the two values differ, the invalid OID (0999 0002) alarm is issued

The system group of RFC1213 contains a *sysLocation* variable that could be used to identify the hardware instances; however, this variable can be (and sometimes is) left undefined. The generic profile polls this variable and, if it is defined, uses it in the same way other cartridges use device specific MIB variables recording the device serial number, that is

- the variable is initially read and stored in the device cache
- subsequently, this variable is polled and if new values are found for both this variable and the device name, the *new hardware* SET alarm (0999 0003) is issued with a CRITICAL severity; however, if only the value of this variable changes, the alarm issued has a WARNING severity and the new value replaces the old one in the cache

If *sysLocation* is not defined then *new hardware* verification is not done and IP address duplication between the device and another with the same system OID is not detected.

Generic DCD configuration files

Four configuration files are required by the GenDCD process using the generic device profile:

- “genericdcd.cfg” (page 265)
- “generic.agp” (page 266)
- “generic.pol” (page 267)
- “generic.tra” (page 268)

The configuration files are included in the Preside Multiservice Data Manager (MDM) fault base, and therefore are directly available to MDM users. The generic device profile defined can be easily customized to modify GenDCD behavior as to

- how the device and interface names are obtained
- how the Cold and Warm Start traps are handled
- which raw state is associated with an interface disabled by the network operator

See “Device behavior options” (page 258) for information on how the generic device profile can be tailored to meet customer requirements.

genericdcd.cfg

This file defines the run-time options applying to the GenDCD process using the generic device profile. The file default version is:

/opt/MagellanNMS/lib/cfg/dcd/genericdcd.cfg

It contains the following lines that specify that this GenDCD instance uses the generic device profile and behavior options with their default values.

```
agentProf: generic

# device name option, the supported values are:
# SYSNAME    - use the device sysName variable (default)
# IPADDRESS  - use the device address
# HOSTNAME   - use the network host name
# COMMAND    - use the name supplied in the discovery
#              command
Cache: NAME_SOURCE:SYSNAME

# interface raw state calculation option,
# the supported values are:
# YES - raw state is OOS if interface disabled by operator
# NO  - raw state is UNK if interface disabled by operator
Cache: OOS_FOR_ADMIN_DOWN:YES

# issue SET alarm for all restarts option, the supported
#         values are:
# YES - SET alarms will be issued for isolated traps
# NO  - SET alarms will only be issued for repeated traps
Cache: ALARM_ALL_RESTARTS:NO
```

generic.agp

This file defines the run-time options applying to the generic SNMP device profile. The default version of this file is

/opt/MagellanNMS/lib/cfg/dcd/generic.agp

It contains the following lines:

```
# select the agent type labeled “generic” in the .pol file
agentType: generic
# use 1 as the device type
deviceType: 1
# use “GEN” as the type for the top level component
nodeType: GEN
# accept any trap
oidFilter: 1.3.6.*
# enable multiple trap addresses support
multiTrapAddr: TRUE
# ignore port number encoded in traps if any
portOverride: TRUE
# declare the SECURITY component as dynamic
dynamicType: SECURITY
# provide IP Discovery with a sysObjectID
# applying to any device
#  sysObjectID:   .1.3.6
```

generic.pol

This file defines the polling requests used by the generic profile device. This file is located at:

/opt/MagellanNMS/lib/cfg/dcd/generic.pol

Note: This file is not intended to be modified by customers.

generic.tra

This file defines the trap translation rules used by the generic profile. This file is located at

/opt/MagellanNMS/lib/cfg/dcd/generic.tra

All the translation records in this file have a SNMPv1 and a SNMPv2c version.

Only the ColdStart, WarmStart, LinkDown, LinkUp and AuthenticationFailure traps are independently supported. The alarms generated are described below and are the same as the alarms generated by most cartridges for these traps. All other traps are translated by the same default translation record generating a MESSAGE alarm containing a comment text attribute reporting the trap OID and specific code for SNMPv1 traps or the trap notification OID for SNMPv2 traps.

In addition, this file contains translation records for internally generated traps:

- invalid system OID detected
- new hardware detected
- ColdStart trap accumulation
- WarmStart trap accumulation
- AuthenticationFailure trap accumulation

ColdStart trap

This trap is translated into a CLEAR alarm against the device with fault code C0000000 and has a hierarchical scope; that is, all of the device's currently active alarms are cleared by this alarm. The trap also causes the device to be rediscovered.

This trap also contributes to trap accumulation record that causes SET alarms to be issued with the same fault code and an increasing severity if the trap is repeated during a 30-minute interval. The severity of the alarm depends on the repetition count in this 30-minute interval as follows:

- two traps in 30 minutes: minor severity
- three traps in 30 minutes: major severity

- four traps or more in 30 minutes: critical severity

Additionally, if the ALARM_ALL_RESTARTS process cache flag is set to YES, the first trap in the repetition sequence also causes a SET alarm with a warning severity to be issued.

The trap accumulation feature also clears any SET alarm issued if no additional trap has been received for 30 minutes.

WarmStart trap

This trap is always translated into a CLEAR alarm against the device with fault code C0000001 and has a hierarchical scope; that is, all of the device's currently active alarms are cleared by this alarm. This trap also causes a new state polling cycle.

This trap also contributes to a trap accumulation record that causes SET alarms to be issued with the same fault code and an increasing severity if the trap is repeated during a 30-minute interval.

- two traps in 30 minutes: minor severity
- three traps in 30 minutes: major severity
- four traps or more in 30 minutes: critical severity

Additionally, if the ALARM_ALL_RESTARTS process cache flag is set to YES, the first trap in the repetition sequence also causes a SET alarm with a warning severity to be issued.

The trap accumulation feature also clears any SET alarm issued if no additional trap has been received for 30 minutes.

LinkDownStart trap

If the OOS_FOR_ADMIN_DOWN process cache flag is set to YES, this trap is translated into a SET alarm with a critical severity and a C0000002 fault code against the corresponding interface component.

If this flag is set to NO, an alarm is not generated, but a state polling request for the corresponding interface table entry is scheduled. The data contained in the reply can cause an alarm to be issued against this interface.

LinkUp trap

This trap is always translated into a CLEAR alarm against the corresponding IF component. If the component was in *UNK* state because the “no alarm if administratively disabled” option is selected, this CLEAR alarm returns it to the *INSV* raw state.

AuthenticationFailure trap

This alarm is always translated into a MESSAGE alarm with a C0000004 fault code against the instance-less SECURITY component.

This trap also contributes to a trap accumulation record that causes a SET alarm to be issued with the same fault code and an increasing severity if the trap is repeated during a 10-minute interval. The severity of the alarm depends on the repetition count in this 10-minute interval as follows:

- three traps in 10 minutes: warning severity
- six traps in 10 minutes: minor severity
- nine traps in 10 minutes: major severity
- 12 traps or more in 10 minutes: critical severity

The trap accumulation feature also clears any SET alarm issued if only one trap has been received in 10 minutes.

Configuring generic SNMP devices

You can configure a generic SNMP device with the SNMP Surveillance Adapter so that traps are monitored and viewed through Preside Multiservice Data Manager (MDM) Fault toolset.

Prerequisites

For information on when and why you would use the generic device profile, see “Generic SNMP device profile” (page 254) and “Generic device model” (page 257).

Procedure steps

1. Perform the following tasks on the device:
   a. Verify that the SNMP device meets the minimum requirements to be managed with the generic profile (see “Minimum and optional requirements” (page 256)).
   b. Configure the device’s SNMP agent to send traps to each workstation where a DCD will be running unless the device supports the MIB required for IP discovery to take care of this task.
   c. If the *sysName* is used to derive the device name, define it in the device MIB.
   d. If protection against duplicate IP addresses is required, define *sysLocation* in the device MIB.
   e. If mnemonic naming of interfaces is required and the device SNMP agent supports the interface table extension, define the corresponding *ifName* entries in the device MIB.
2. Modify the configuration files, if necessary. See “Modifying configuration files” (page 272).
3. Start the GenDCD process. See “Starting the GenDCD process” (page 273).
4. Using the IP Discovery tool, trigger the discovery of the device. See “IP discovery” (page 29).

Modifying configuration files

The Generic (SNMP) Data Collection Daemon (GenDCD) process is compatible with the way Preside Multiservice Data Manager handles non-SNMP devices. To modify the defaults, changes can be made in the *genericdcd.cfg* or *generic.agp* files to add the lines required to select the non-default behavior.

If the modifications apply to all GenDCD instances using the generic profile on the same workstation, modify the *generic.agp*. If separate GenDCD instances using the generic profile have different behavior, modify the respective *.cfg* file. See “Generic DCD configuration files” (page 264) and “Device behavior options” (page 258).

Procedure steps

1. Copy the default file version required from /opt/MagellanNMS/lib/cfg/dcd to /opt/MagellanNMS/cfg/dcd.
2. Add the lines required using a text editor.

Starting the GenDCD process

The GenDCD process must be started since it is not associated with the installation script of a corresponding cartridge.

Prerequisites

The SMDR process to which the new GenDCD process is reporting must be running. See the 241-6001-310 *Preside MDM Server Reference Guide* for information on how to create a SMDR server and connect it to the GMDR processes.

Procedure steps

1. Start the Server Administration tool. See the 241-6001-303 *Preside MDM Administrator Guide*.
2. Add a new GenDCD server with the following command line:

   ```
   /opt/MagellanNMS/bin/gendcd -N generic
   ```

3. On the workstation on which SMDR is running, use the *smdrCreateServer* command to connect SMDR to the new GenDCD process. See the 241-6001-303 *Preside MDM Administrator Guide*.

Appendix B
Repeater examples

This section contains examples of device model and file designs for a repeater device using the SNMP Surveillance Adapter. It contains the following topics:

About repeaters

A repeater is a multiple ports device that repeats on all the other ports any input received on one of those ports; it is also often called a Hub. No address-based routing is performed. The ports are not considered independent interfaces because they do not have individual addresses and MIB-II interface table is not supported. Repeaters were initially supported by RFC 1368, which was subsequently replaced by RFC 1516 and later replaced by RFC 2108.

RFC 1516

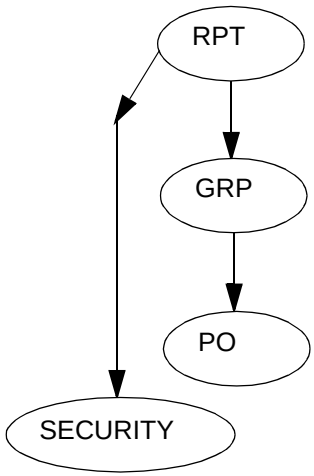
RFC 1516 has the following characteristics:

- RFC 1516 models the device with a set of port groups. The group table defines each group. The port table defines all the ports from all the groups.
- The port grouping mechanism is left unspecified. It could, for example, correspond to separate physical boards.
- Only one fault signaling trap is defined: repeater health trap. When group or port failure is signaled, the instance is not identified; this is a major design problem. The trap is sent only if device overall operational status changes. For example, a second port failure is not signaled if it occurs while the repeater is already in port failure state.

Therefore trap flow is insufficient to monitor the device. You must rely on polling with more frequent state polling cycles.

Repeater device model

The following figure is a hierarchical representation of the device model for a repeater.



This device model consists of three layers:

- top component (device)
  - name from sysName (1213)
  - state from rptrOperStatus (1516)
- port group
  - component instance from rptrGroupIndex (1516)
  - state from rptrGroupOperStatus (1516)
- port
  - component instance from rptrPortIndex (1516)
  - state from rptrPortAdminStatus, rptrPortAutoPartitionState, rptrPortOperStatus (1516)
- security
  - component instance: $
  - dynamic component used to report “security” alarms
  - not polled

Network model configuration

Three configuration files are required to add this device model to the Preside Multiservice Data Manager (MDM) Network Model schema:

- RPT_types.mtdf specifies the component hierarchy for this device model.
- RPT_types.otdf specifies in which network model organizations this type of device can occur.

  Note: This file is no longer required for MDM release versions 12.5 or higher.

- RPT.xpm defines the device icon.

These files are installed as follows:

- .mtdf and .otdf: /opt/MagellanNMS/ext/lib/model/types

- .xpm: /opt/MagellanNMS/ext/lib/nds/pixmaps

  Note: The .ltdf and .atdf configuration files are not required because we are not introducing new link or attribute types.

RPT_types.mtdf

The following declarations define repeater module and subcomponent types. This file contains declarations for the device category (RPT), port group category (GRP), port category (PO), and security respectively.

```
Module_type: RPT
    Legal_name: ..*
    Label: Repeater
    Verbatim_start Explanation:
      Repeaters are identified by
      their module name (string).
    Verbatim_end:
    Flag: BACKBONE
    Criticality: 4
    Info_attribute: INFO
    Info_attribute: NODE_PIXMAP RPT.xpm

Subcomponent_type: GRP
    In_module: RPT
    Parent: RPT
    Legal_name: ..*
    Label: Repeater port group
    Verbatim_start Explanation:
      The instance of a Repeater Group
      is an integer (rptrGroupIndex)
    Verbatim_end:
    Subtype: ORDINARY
    Criticality: 3

Subcomponent_type: PO
    In_module: RPT
    Parent: GRP
    Verbatim_start Explanation:
      The instance of a Repeater Port
      is an integer (rptrPortIndex)
    Verbatim_end:
    Legal_name: ..*
    Label: Repeater port
    Subtype: ENDPOINT
    Criticality: 2

Subcomponent_type: SECURITY
    In_module: RPT
    Parent: RPT
```

```
    Verbatim_start Explanation:
      The SECURITY component has
      no instance
    Verbatim_end:
    Legal_name: ..*
    Label: Device security
    Subtype: DYNAMIC
    Criticality: 3
```

RPT_types.otdf

Note: This file is no longer required for MDM release versions 12.5 or higher.

The following declarations allow repeater devices in all the organizations defined in /opt/MagellanNMS/ext/lib/model/types/types.otdf

```
Organization_type: GENERIC
    Module_types: RPT

Organization_type: DEFAULT
    Module_types: RPT

Organization_type: MAGELLAN
    Module_types: RPT

... (other organizations)
```

RPT.xpm: device icon definition

The following declarations define the icon used in the Network Viewer.

```
/* XPM */
static char * RPT_xpm[] = {
/* width height ncolors cpp [x_hot y_hot] */
"21 21 8 1 0 0",
/* colors */
"        s none m none c none",
"        s select Color m white c #737373737373",
"X       s background   m black c #949494949494",
"o       s iconColor1   m black c black",
"O       s bottomShadowColor m black c #636363636363,
"+       s iconGray6    m black c #636363636363"
"@       s topShadowColor m white c #bdbdbdbdbdbd",
"#       s iconGray8      m black c #212121212121",
/* pixels */
"                    ",
"   ..................",
"  ..................",
" ....XXXXXXXXXXXXX..",
"....XoOOOOOOOOOOX+..",
"...X+oOOOOOOOOOX++..",
"..XXXXXXXXXXXXX+++..",
"..X@@@@@@@@@@@#++++..",
"..X@@@@@@@@@@@@++++..",
"..X@@@#######@@@++++..",
"..X@@@@@@@@@@@@++++..",
"..X@@@@@@@@@@@#++++..",
"..X@@@#@@@##########++..",
"..X@@@##@@##########++..",
"..X@@@########@#######++..",
"..X@@@########@#######++..",
"..X@@@#######@@@#+++...",
"..X@@@#######@@@++....",
"..O++++++++++++.... ",
"..................  ",
".................   "};
```

SNMP Surveillance Adapter configuration

To configure a generic DCD process to monitor repeaters, four configuration files are required:

- "<process name>.cfg file" (page 282) specifies process global parameters
- "rptr.agp file" (page 283) specifies configuration parameters specific to repeaters
- "rptr.pol file" (page 283) defines which polling requests to send to repeaters and how to handle responses
- "rptr.tra file" (page 297) specifies how traps received from repeaters are translated into alarms

Note: These files could be installed in /opt/MagellanNMS/ext/lib/cfg/dcd

<process name>.cfg file

The only options required in the process options configuration file are the following:

- an address filter (if the surveillance network deployment requires it)
- the list of profiles supported

The process run-time options configuration file should define the following parameters:

```
addrFilter: 45.136.30-50.* #Montreal region
agentProf: rptr
```

OR

```
addrFilter: 45.136.30.50/12 #Montreal region
agentProf: rptr
```

Note: Default values are acceptable for the other parameters.

rptr.agp file

The following example shows an agent profile (.agp) configuration file.

```
nodeType: RPT
deviceType: 51
oidFilter: 1.3.6.1.2.1.22  #snmpDot3RptrMgt OID
oidFilter: 1.3.6.1.2.1     # for generic traps
maxStPollInt: 120     # refresh states every 2 minutes
agentType: rptr
```

This .agp configuration file contains the following specifications:

- The device type (first token of the component identifiers) assigned to those devices is "RPT".
- The device type assigned to the specified devices is 51. This device type value is used in seed files, the dcdAddNode script, alarm fault codes, and other areas.
- Two OID filter elements are defined for this profile:
  - The first is the object identifier of the branch defined by RFC 1516 which will match the OID of traps defined by this MIB.
  - The second is the MIB-II OID used to match generic traps OID.
- Because the device trap flow is insufficient, faster state polling intervals are specified (2 minutes instead of 5).
- The agent type is "rptr"; this is the name of the corresponding TYPE declaration in the .pol configuration file.

rptr.pol file

The CLASS declaration of a polling (.pol) configuration file

- defines the polling requests required
- specifies how to handle responses

The TYPE declaration of a .pol configuration file enumerates the polling requests associated with the agent type.

Device polling configuration

This section provides the MIB definitions and the corresponding device discovery and device state declarations.

MIB definitions

```
rptrOperStatus OBJECT-TYPE
 SYNTAX INTEGER {
    other(1), -- undefined or unknown status
    ok(2), -- no known failures
    rptrFailure(3), -- repeater-related failure
    groupFailure(4), -- group-related failure
    portFailure(5), -- port-related failure
    generalFailure(6) -- failure, unspecified type
}
DESCRIPTION
  "The rptrOperStatus object indicates the operational
   state of the repeater.The rptrHealthText object may
   be consulted for more specific information about the
   state of the repeater's health. In the case of
   multiple kinds of failures (e.g., repeater failure
   and port failure), the value of this attribute shall
   reflect the highest priority failure in the
   following order, listed highest priority first:
   rptrFailure(3) groupFailure(4) portFailure(5)
   generalFailure(6)."
::={rptrRptrInfo 2}
```

Device discovery

```
  CLASS rptrNodeDisc
  {
1  REQGROUP 0
2  COMPTYPE 0
3  METHOD single

4  sysName      1.3.6.1.2.1.1.5.0
5  operStatus    1.3.6.1.2.1.22.1.1.2.0

6  CompId: [EMPTY(sysName) ? DEVNAME() \
           MAKENAME(sysName)]
7  State: [EQ(operStatus, 2) ? INSV \
           OR( EQ(operStatus, 4), EQ(operStatus, 5))\
           ? ISTB
           OR( EQ(operStatus, 3), EQ(operStatus, 6))\
           ? OOS \
           UNK ]
   }
```

Note: Some examples in this appendix contain numbers at the start of each line. These numbers are not part of the actual request; they have been added for clarity purposes to refer to individual lines.

This device discovery example is the polling request declaration contained in the configuration file rptr.pol that is used to discover the device component itself.

Lines 1 to 3 define the request attributes:

- Line 1: the request declared is in the discovery group
- Line 2: the request is associated with the component type 0 (device)
- Line 3: an SNMP GET request is used

Lines 4 to 5 define the polled variables:

- Line 4: the sysName variable (RFC 1213) is polled for the device name
- Line 5: the rptrOperStatus variable (RFC 1516) is polled for the device overall state

Lines 6 to 7 specify the commands to execute to handle responses to this request:

- Line 6: the device component identifier is computed as follows:
  - — If the system name is not defined, continue using the name initially assigned to the device by the dcdAddNode request or by trap based discovery; otherwise, use a sanitized version of the system name.
- Line 7: the device state is defined as InService, InServiceTroubled or OutOfService depending on the polled device operational status

Device state

```
     CLASS rptrNodeState
     {
1    REQGROUP 1
2    COMPTYPE 0
3    METHOD single

4    operStatus   1.3.6.1.2.1.22.1.1.2.0

5    CompId: DEVNAME()
6    State: [EQ(operStatus, 2) ? INSV \
             OR(EQ(operStatus, 4), EQ(operStatus, 5))\
             ? ISTB \
             OR(EQ(operStatus, 3), EQ(operStatus, 6)) \
             ? OOS \
             UNK]
      }
```

This device state example is the polling request declaration contained in the configuration file rptr.pol. It is used to verify the state of the device component itself.

Lines 1 to 3 define the request attributes:

- Line 1: the request declared is in the state group
- Line 2: the request is associated with the component type 0 (device)
- Line 3: an SNMP GET request is used

Line 4 defines the only polled variable which is the rptrOperStatus variable (RFC 1516) polled for the device overall state.

Lines 5 to 6 specify the commands to execute to handle responses to this request:

- Line 6: the device component identifier is defined as the device name currently associated with the polled object
- Line 7: the device state is defined as InService, InServiceTroubled or OutOfService depending on the polled device operational status

Group component polling configuration

This section provides the MIB definitions and the corresponding group discovery and group state declarations.

MIB definitions

```
rptrGroupIndex OBJECT-TYPE
  SYNTAX INTEGER (1.. 1024)
  DESCRIPTION
  "This object identifies the group within the repeater
  for which this entry contains information. This value
  is never greater than rptrGroupCapacity."
 ::= {rptrGroupEntry 1 }

rptrGroupOperStatus OBJECT-TYPE
  SYNTAX INTEGER { other(1), operational(2),
                   malfunctioning(3), notPresent(4),
                   underTest(5), resetInProgress(6) }
  DESCRIPTION
   "An object that indicates the operational status of
   the group. A status of notPresent(4) indicates that
   the group is temporarily or permanently physically
   and/or logically not a part of the repeater. It is
   an implementation-specific matter as to whether the
   agent effectively removes notPresent entries from
   the table. A status of operational(2) indicates that
   the group is functioning, and a status of
   malfunctioning(3) indicates that the group is
   malfunctioning in some way."
::={rptrGroupEntry 4}
```

Group discovery

```
     CLASS rptrGroupDisc
     {
1    REQGROUP 0
2    COMPTYPE 1
3    METHOD table

4    groupIndex        1.3.6.1.2.1.22.1.2.1.1.1
5    groupOperStatus   1.3.6.1.2.1.22.1.2.1.1.5

6    Quit: EQ(groupOperStatus, 4)
7    Assign: name SPCAT( DEVNAME(), "GRP", groupIndex)
8    CompId: name
9    Cache: SPCAT("GROUP", groupIndex) name
10   State: [ EQ(groupOperStatus, 2) ? INSV \
              OR( EQ(groupOperStatus, 3), \
                  EQ(groupOperStatus, 5), \
                  EQ(groupOperStatus, 6)) ? OOS \
              UNK ]
     }
```

This group discovery example is the polling request declaration contained in the configuration file rptr.pol. It is used to discover the port group components.

Lines 1 to 3 define the request attributes:

- Line 1: the request declared is in the discovery group
- Line 2: the request is associated with the component type 1
- Line 3: an SNMP GETNEXT request is used

Lines 4 to 5 define the polled variables:

- Line 4: the rptrGroupIndex variable is polled for the group component instance
- Line 5: the rptrGroupOperStatus variable is polled for the group overall state

Lines 6 to 10 specify commands executed to handle responses to this request:

- Line 6: if the operation status is 4 (absent), the response is discarded because the group does not really exist
- Line 7: the string obtained by appending the token GRP and the value of the group index to the device name is assigned to the local variable name
- Line 8: the component identifier is defined as the value of name
- Line 9: the value of name is cached in a cache entry whose name is “GROUP” followed by the group index; this cache entry contains the component identifier retrieved later to process responses to the corresponding state requests
- Line 10: the component state is computed from the operational status

Group state

```
     CLASS rptrGroupState
     {
1    REQGROUP 1
2    COMPTYPE 1
3    METHOD table

4    groupIndex        1.3.6.1.2.1.22.1.2.1.1.1
5    groupOperStatus   1.3.6.1.2.1.22.1.2.1.1.5

6    Quit: EQ(groupOperStatus, 4)
7    Assign: name CACHE(SPCAT("GROUP", groupIndex))
8    Discover: EMPTY(name)
9    CompId: name
10   State: [ EQ(groupOperStatus, 2) ? INSV \
              OR( EQ(groupOperStatus, 3), \
                  EQ(groupOperStatus, 5), \
                  EQ(groupOperStatus, 6)) ? OOS \
              UNK ]
     }
```

This group state example is the polling request declaration contained in the configuration file rptr.pol. It is used to verify the state of the port group components.

Lines 1 to 3 define the request attributes:

- Line 1: the request declared is in the state group
- Line 2: the request is associated with the component type 1
- Line 3: an SNMP GETNEXT request is used

Lines 4 to 5 define the polled variables.

- Line 4: the rptrGroupIndex variable is polled for the group component instance
- Line 5: the rptrGroupOperStatus variable is polled for the group overall state

Lines 6 to 10 specify the commands to execute to handle responses to this request:

- Line 6: if the operation status is 4 (absent), the response is discarded because the group does not really exist
- Line 7: the string obtained by searching the cache for an entry whose name is GROUP followed by the group index is assigned to the local variable name
- Line 8: if the cache search failed, name contains an empty string which means this response is dealing with a newly configured group. A new discovery polling cycle is scheduled and the response is discarded.
- Line 9: the component identifier is defined as the value of name
- Line 10: The component state is computed from the operational status

Port component polling configuration

This section provides the MIB definitions and the corresponding group discovery and group state declarations.

MIB definitions

```
rptrPortIndex OBJECT-TYPE
   SYNTAX INTEGER (1.. 1024)
   DESCRIPTION
    "This object identifies the port within the group
    for which this entry contains information. This
    value can never be greater than
    rptrGroupPortCapacity for the associated group."
::= { rptrPortEntry 2}

rptrPortAutoPartitionState OBJECT-TYPE
   SYNTAX INTEGER { notAutoPartitioned(1),
   autoPartitioned(2) }
   DESCRIPTION
    "The autoPartitionState flag indicates whether the
    port is currently partitioned by the repeater's
    auto-partition protection. The conditions that
    cause port partitioning are specified in partition
    state machine in Section 9 [IEEE 802.3 Std]. They
    are not differentiated here."
::= { rptrPortEntry 4}

rptrPortAdminStatus OBJECT-TYPE
   SYNTAX INTEGER { enabled(1), disabled(2) }
   DESCRIPTION
    "Setting this object to disabled(2) disables the
    port. A disabled port neither transmits nor
    receives. Once disabled, a port must be
    explicitly enabled to restore operation. A port
    which is disabled when power is lost or when a
    reset is exerted shall remain disabled when normal
    operation resumes. The admin status takes
    precedence over auto-partition and functionally
    operates between the auto-partition mechanism and
    the AUI/PMA. Setting this object to enabled(1)
    enables the port and exerts a BEGIN on the port's
    auto-partition state machine. (In effect, when a
    port is disabled, the value of
```

```
        rptrPortAutoPartitionState for that port is
        frozen until the port is next enabled. When the
        port becomes enabled, the
        rptrPortAutoPartitionState becomes
        notAutoPartitioned(1), regardless of its pre-
        disabling state.)"
    ::= { rptrPortEntry 3 }

    rptrPortOperStatus OBJECT-TYPE
       SYNTAX INTEGER { operational(1), notOperational(2),
        notPresent(3) }
       DESCRIPTION
        "This object indicates the port's operational
        status. The notPresent(3) status indicates the
        port is physically removed (note this may or may
        not be possible depending on the type of port.) The
        operational(1) status indicates that the port is
        enabled (see rptrPortAdminStatus) and working,
        even though it might be auto-partitioned (see
        rptrPortAutoPartitionState). If this object has
        the value operational(1) and rptrPortAdminStatus
        is set to disabled(2), it is expected that this
        object's value will soon change to
        notOperational(2)."
    ::= { rptrPortEntry 5}
```

Port discovery

```
     CLASS rptrPortDisc
     {
1    REQGROUP 0
2    COMPTYPE 2
3    METHOD table

4    groupIndex           1.3.6.1.2.1.22.1.3.1.1.1
5    portIndex            1.3.6.1.2.1.22.1.3.1.1.2
6    portOperStatus       1.3.6.1.2.1.22.1.3.1.1.5
7    portAdmStatus        1.3.6.1.2.1.22.1.3.1.1.3
8    portPartitionSt      1.3.6.1.2.1.22.1.3.1.1.4

9    Quit: EQ(portOperStatus, 3)
10   Assign: name SPCAT( DEVNAME(), "GRP", groupIndex,\
                         "PO", portIndex)
11   CompId: name
12   Cache: SPCAT("PORT", groupIndex, portIndex) name
13   State: [ AND( EQ( portPartitionSt, 1), \
                   EQ (portOperStatus, 1), \
                   EQ(portAdmStatus, 1) ) ? INSV \
               OOS ]
     }
```

This port discovery example is the polling request declaration contained in the configuration file rptr.pol. It is used to discover the port components.

Lines 1 to 3 define the request attributes:

- Line 1: the request declared is in the discovery group
- Line 2: the request is associated with the component type 2
- Line 3: an SNMP GETNEXT request is used

Lines 4 to 8 define the polled variables:

- Lines 4 to 5: the group and port index variables (RFC 1516) are polled for the group and port component instances
- Lines 6 to 8: the operational, administrative and partition status variables (RFC 1516) are polled for the port state

Lines 9 to 13 specify the commands to execute to handle responses:

- Line 9: if the operation status is 3 (absent), the response is discarded because the port does not really exist
- Line 10: the string obtained by appending the token GRP, the group index, the token PO and the port index to the device name is assigned to name
- Line 11: the component identifier is defined as the value of name
- Line 12: the value of name is cached in a cache entry whose name is “PORT” followed by the group and the port indices; therefore, this cache entry contains the component identifier which will be retrieved later to process the responses to the corresponding state requests
- Line 13: the component state is computed from the status variables

Port state

```
     CLASS rptrPortState
     {
1    REQGROUP 1
2    COMPTYPE 2
3    METHOD table

4    groupIndex         1.3.6.1.2.1.22.1.3.1.1.1
5    portIndex          1.3.6.1.2.1.22.1.3.1.1.2
6    portAdmStatus      1.3.6.1.2.1.22.1.3.1.1.3
7    portOperStatus     1.3.6.1.2.1.22.1.3.1.1.5
8    portPartitionSt    1.3.6.1.2.1.22.1.3.1.1.4

9    Quit: EQ(portOperStatus, 3)
10   Assign: name CACHE( SPCAT( "PORT", groupIndex,\
                                portIndex))
11   Discover: EMPTY(name)
12   CompId: name
13   State: [ AND( EQ(portPartitionSt, 1), \
                   EQ(portOperStatus, 1), \
                   EQ(portAdmStatus, 1) ) ? INSV \
              OOS ]
     }
```

This port state example is the polling request declaration contained in the configuration file rptr.pol. It is used to verify the state of the port components.

Lines 1-3 define the request attributes:

- Line 1: the request declared is in the state group
- Line 2: the request is associated with the component type 2
- Line 3: an SNMP GETNEXT request is used

Lines 4-8 define the polled variables:

- Lines 4 to 5: the group and port index variables are polled for the group and port component instances
- Lines 6 to 8: the operational, administrative, and partition status variables are polled for the port state

Lines 9 to 13 specify the commands to execute to handle respones to this request:

- Line 9: if the operation status is 3 (absent), the response is discarded because the port does not really exist
- Line 10: the string obtained by searching the cache for an entry whose name is PORT, followed by the group index and the port index is assigned to name
- Line 11: if the cache search failed, name contains an empty string which means this response is dealing with a newly configured port. A new discovery polling cycle is scheduled and the response is discarded.
- Line 12: the component identifier is defined as the value of name
- Line 13: the component state is computed from the status variables

Other polling configuration file declarations

This section provides examples of reachability and agent type declarations.

Reachability

```
     CLASS rptrReachability
     {
1    REQGROUP 2
2    COMPTYPE 0
3    METHOD single

4    sysName      1.3.6.1.2.1.1.5.0

5    CompId: [ EMPTY(sysName) ?  DEVNAME() \
               MAKENAME(sysName)]
     }
```

This example is the polling declaration contained in the configuration file rptr.pol that is used to verify the reachability of the device. The fact that a response is received already establishes the device reachability. In this example, the request is also used to detect device name changes.

Lines 1 to 3 define the request attributes:

- Line 1: the request declared is in the reachability group
- Line 2: the request is associated with the component type 0
- Line 3: an SNMP GET request is used

Line 4 defines the only polled variable: sysName

Line 5 recomputes the device name based on sysName contents; if a change of name is detected, a new discovery polling cycle is scheduled.

Agent type

The agent type declaration is included in the polling configuration file.

```
TYPE rptr
{
  rptrNodeDisc
  CLASSNAME rptrGroupDisc
  CLASSNAME rptrPortDisc
  CLASSNAME rptrNodeState
  CLASSNAME rptrGroupState
  CLASSNAME: rptrPortState
  CLASSNAME rptr Reachability
}
```

This declaration lists all the polling request class names associated with the specified agent type.

The name of the agent type is the token following the TYPE keyword. This agent type name is associated with an agent profile in the .agp configuration file.

rptr.tra file

The trap translation (.tra) file specifies how to translate each trap into an alarm.

Cold and warm start traps

```
1   Trap_oid: 1.3.6.1.2.1
2   Trap_code: -1, -2
3   Comp_id: E DEVNAME()
4   Fault_code: E [EQ(GENERIC(), 0) ? "C0000000" \
                  "C0000001"]
5   Severity: S cleared
6   Event: S clear
7   AlmType: E [EQ(GENERIC(), 0) ? "equipment" \
                "operator"]
8   ProbCause: E [EQ(GENERIC(), 0) ? \
                  "equipmentFailure" \
                  "operationalConditions"]
9   Comment: E[EQ(GENERIC(), 0) ? "Cold start trap" \
              "Warm start trap"]
10   ClrScope: S clearHier
11   Discover: E TRUE
```

This trap translation record contained in the rptr.tra configuration file defines the translation rules for the cold and warm start traps.

- Line 1: the trap object identifier (OID) is the RFC 1213 OID
- Line 2: two trap codes are associated with this translation record; negative trap codes identify the traps as generic traps by the rule (code = -(generic+1))
- Line 3 defines the alarm component identifier as the device name
- Line 4 selects the alarm fault code based on the trap generic code
- Line 5 sets the alarm severity as “cleared”
- Line 6 sets the alarm event type as CLEAR
- Line 7 selects the alarm type based on the trap generic code
- Line 8 selects the alarm probable cause based on the trap generic code
- Line 9 selects the alarm comment based on the trap generic code
- Line 10 specifies that this alarm is intended to clear all alarms currently active against the device and its components
- Line 11 specifies that this alarm causes a new discovery polling cycle to be scheduled

Authentication failure trap

```
1   Trap_oid: 1.3.6.1.2.1
2   Trap_code: -5
3   Comp_id: E SPCAT(DEVNAME(), “SECURITY $”)
4   Fault_code: S C0000004
5   Severity: S major
6   Event: S set
7   AlmType: S security
8   ProbCause: S authenticationFailure
9   Comment: S Authentication failure
10   Age: 1
```

SECURITY component justification:

Authentication failures are significant events; the network operator must be made aware of them. However, raising the alarms against the device component is not satisfactory since the device itself is not malfunctioning in any manner.

This trap translation record contained in the rptr.tra configuration file defines the translation rules for the authentication failure trap.

- Line 1: the trap OID is the RFC 1213 OID
- Line 2 associates the authentication failure trap code with this translation record. The negative trap code identifies the trap as a generic trap by the rule
  (code = -(generic+1))
- Line 3 defines component impacted by the alarm as the SECURITY component under the device
- Line 4 defines the alarm fault code as C0000004
- Line 5 sets the alarm severity as “major”
- Line 6 sets the alarm event type as “set”
- Line 7 selects “security” as the alarm type
- Line 8 selects authenticationFailure as the alarm probable cause
- Line 9 defines the alarm comment
- Line 10 specifies that the alarm must be cleared if it is still active after one hour

Health trap

This section provides the MIB definition and trap translation record for a health trap.

MIB definition

```
rptrHealth TRAP-TYPE
  ENTERPRISE snmpDot3RptrMgt
  VARIABLES { rptrOperStatus }
  DESCRIPTION
   "The rptrHealth trap conveys information related to
   the operational status of the repeater. This trap
   is sent either when the value of rptrOperStatus
   changes, or upon completion of a non-disruptive
   test. The rptrHealth trap must contain the
   rptrOperStatus object. The agent may optionally
   include the rptrHealthText object in the varBind
   list. See the rptrOperStatus and rptrHealthText
```

```
        objects for descriptions of the information that is
        sent. The agent must throttle the generation of
        consecutive rptrHealth traps so that there is at
        least a five-second gap between traps of this type.
        When traps are throttled, they are dropped, not
        queued for sending at a future time. (Note that
        "generating" a trap means sending to all configured
        recipients.)"
    ::=1
```

Example

```
1   Trap_oid: 1.3.6.1.2.1.22
2   Trap_code: 1
3   Comp_id: E DEVNAME()
4   Fault_code: S C0510001
5   Name: state 1.3.6.1.2.1.22.1.1.2
6   Severity: E [ EQ(state, 2) ? "cleared" "major"]
7   Event: E[ EQ(state, 2) ? "clear" "set" ]
8   AlmType: S communications
9   ProbCause: S outOfService
10  Comment: E VAR(1.3.6.1.2.1.22.1.1.3)
11  Poll: E TRUE
```

This trap translation record defines the translation rules for the RFC 1516 repeater health trap.

- Line 1: the trap OID is the RFC 1516 OID
- Line 2: the trap code identifies the trap specific code
- Line 3: the alarm is raised against the device itself even when it signals a port failure because the trap does not identify any subcomponent
- Line 4: the fault code identifies the device type (“051”) and assigns “0001” to the problem reported by the health trap
- Line 5: the name state is associated with the trap variable reporting the new repeater state
- Lines 6 to 7: a CLEAR alarm is generated if the state variable has the value 2 (ok); otherwise a “major” SET alarm is generated
- Lines 8 to 9: the alarm type and probable cause attributes are hard-coded to communications and outOfService respectively

- Line 10: the comment attribute is directly copied from the corresponding trap variable
- Line 11: a new state poll cycle is scheduled to establish the impact of the problem reported

Group change trap

This section provides an example of a MIB definition and trap translation record for a group change trap.

MIB definition

```
rptrGroupChange TRAP-TYPE
  ENTERPRISE snmpDot3RptrMgt
  VARIABLES { rptrGroupIndex }
  DESCRIPTION
   This trap is sent when a change occurs in the group
   structure of a repeater. This occurs only when a
   group is logically or physically removed from or
   added to a repeater. The varBind list contains the
   identifier of the group that was removed or added.
   The agent must throttle the generation of
   consecutive rptrGroupChange traps for the same
   group so that there is at least a five-second gap
   between traps of this type. When traps are
   throttled,they are dropped, not queued for sending
   at a future time. (Note that "generating" a trap
   means sending to all configured recipients.)"
::= 2
```

Example

```
1   Trap_oid: 1.3.6.1.2.1.22
2   Trap_code: 2
3   Comp_id: E DEVNAME()
4   Fault_code: S C0510002
5   Severity: S warning
6   Event: S message
7   AlmType: S operator
8   ProbCause: S otherOperational
9   Comment E SPCAT("Group", VAR(0), \
                    "has been added or deleted")
10  Discover: E TRUE
```

This trap translation record defines the translation rules for the RFC 1516 repeater group change trap issued when a group is added or deleted.

- Line 1: the trap OID is the RFC 1516 OID
- Line 2: the trap code identifies the trap specific code
- Line 3: the alarm is raised against the device itself because the group component either does not exist anymore or is not yet discovered
- Line 4: the fault code identifies the device type (“051”) and assigns “0002” to the event reported by this trap
- Lines 5 to 6: an MSG alarm with a severity set to “warning” is generated
- Lines 7 to 8: the alarm type and probable cause attributes are hard-coded to operator and other Operational respectively
- Line 9: the trap variable specifying the index of the group affected is included in the alarm comment attribute
- Line 10: a new discovery polling cycle is scheduled

Reset event trap

This section provides an example of a MIB definition and trap translation record for a reset event trap.

MIB definition

```
rptrResetEvent TRAP-TYPE
  ENTERPRISE snmpDot3RptrMgt
  VARIABLES { rptrOperStatus }
  DESCRIPTION
   "The rptrResetEvent trap conveys information
   related to the operational status of the repeater.
   This trap is sent on completion of a repeater reset
   action. A repeater reset action is defined as a
   transition to the START state of Fig. 9-2 in
   section 9 [IEEE 802.3 Std], when triggered by a
   management command (e.g., an SNMP Set on the
   rptrReset object). The agent must throttle the
   generation of consecutive rptrResetEvent traps so
   that there is at least a five-second gap between
   traps of this type. When traps are throttled,
   they are dropped, not queued for sending at a future
   time. (Note that "generating" a trap means sending
```

```
to all configured recipients.) The rptrResetEvent
trap is not sent when the agent restarts and sends
an SNMP coldStart or warmStart trap. However, it is
recommended that a repeater agent send the
rptrOperStatus object as an optional object with its
coldStart and warmStart trap PDUs. The
rptrOperStatus object must be included in the
varbind list sent with this trap. The agent may
optionally include the rptrHealthText object as
well."
::=3
```

Example

```
1   Trap_oid: 1.3.6.1.2.1.22
2   Trap_code: 3
3   Comp_id: E DEVNAME()
4   Fault_code: S C0510003
5   Name: state 1.3.6.1.2.1.22.1.1.2
6   Severity: E [ EQ(state,2) ? “cleared” “major”]
7   Event: E [ EQ(state, 2) ? “clear” “set” ]
8   AlmType: S operator
9   ProbCause: S otherOperational
10  Comment: S Device reset event
11  Poll: E TRUE
```

This trap translation record defines the translation rules for the RFC 1516 repeater reset event trap.

- Line 1: the trap OID is the RFC 1516 OID
- Line 2: the trap code identifies the trap specific code
- Line 3: the alarm is raised against the device itself
- Line 4: the fault code identifies the device type (“051”) and assigns “0003” to the event reported by the reset event trap
- Line 5: the name state is associated with the trap variable reporting the resulting repeater state
- Lines 6 to 7: a CLEAR alarm is generated if the state variable has the value 2 (ok); otherwise a “major” SET alarm is generated
- Lines 8 to 9: the alarm type and probable cause attributes are hard-coded to operator and otherOperational respectively

- Line 10: the comment attribute is hard-coded
- Line 11: a new state poll cycle is scheduled to establish the impact of the event reported

Appendix C
Dispatcher profile examples

This section contains file designs for a dispatcher profile using the SNMP Surveillance Adapter. It contains the following topics:

Note: This appendix contains an example of a dispatcher profile that a DCD would use to manage several types of Cisco devices.

About dispatcher profiles

A dispatcher profile is only used to discover the correct device type of the device at a given address by polling the device system object identifier (OID). When the correct device type is found, the corresponding profile is activated and the dispatcher polling agent structure is destroyed.

The dispatcher profile should be the first profile listed in the global run-time options (.cfg) configuration file. This ensures that it is selected when several profiles could be selected for an incoming trap. The default translation may

be required if a trap is received before the correct device type has been found. This default translation record can simply discard the trap or issue a MESSAGE alarm.

What is a dispatcher profile?

A dispatcher profile should be one of the profiles configured in a data collection daemon (DCD) supporting several device types. A dispatcher profile usually contains only:

- a device discovery request
- a reachability request
- a default trap translation record

Why use a dispatcher profile?

A dispatcher profile finds out through polling which type of SNMP device corresponds to a specified IP address. A dispatcher profile creates a polling agent that corresponds to the device type identified.

Trap-based discovery

Trap-based discovery can be used for traps that are shared by several device types. If the correct profile cannot be identified based on the trap header, the following activities should occur:

1 Dispatcher profile is selected; this will find out which profile applies to the device.

2 The device is unmanaged if it is unsupported.

When trap-based discovery is enabled, some traps can be sent by devices of more than one type supported on the same DCD. When the DCD receives one of these traps from a device that has not been discovered, it cannot decide which device profile to activate. A dispatcher profile is activated and the profile polls the device to find out which profile applies to it.

If, after inspection, the dispatcher profile finds that this DCD does not have the profile required to manage the device, the agent structure should be unmanaged. When an agent is unmanaged, it is not polled. No UDP socket is allocated to it and the traps received from it are discarded. The dispatcher profile puts an agent using it into unmanaged state by selecting a profile that this DCD does not support.

Device type selection errors

There are other circumstances in which a dispatcher profile may be useful. If device polling reveals that the profile being used is not the correct one, activating the dispatcher profile so that it finds the correct profile can solve the problem.

The following are device type selection errors:

- seed file or dcdAddNode specifies the wrong device type
- device is reconfigured so that its device type changes

Network model configuration

A network model configuration is not required for a dispatcher profile because it should never be used to create a component.

SNMP Surveillance Adapter configuration

The SNMP Surveillance Adapter requires configuration to use a dispatcher profile. Four configuration files are required.

Configuration files required

To configure a generic data collection daemon (DCD) process to use a dispatcher profile, the following configuration files are required:

- "<process name>.cfg file" (page 308). This file specifies process global parameters.
- "disp.agp file" (page 309). This file specifies configuration parameters specific to the dispatcher profile.
- "disp.pol file" (page 310). This file defines which polling requests are sent by the dispatcher profile and how to handle responses.
- "disp.tra file" (page 314). This file concerns traps that are received from the device while the dispatcher profile is active. The file specifies how these traps are translated into alarms.

File locations

These configuration files are stored in the normal location, which is one of the following directories:

1 /opt/MagellanNMS/cfg/dcd/
This is the directory for customers to create their files, or store modified copies of default files.

2 /opt/MagellanNMS/ext/lib/cfg/dcd/
This is the directory for other Nortel Networks packages to install their configuration files.

3 /opt/MagellanNMS/lib/cfg/dcd/
This is the directory for Preside Multiservice Data Manager installation from the CD.-ROM

<process name>.cfg file

The only options required in the process run-time options configuration file are the following:

- an address filter (if the surveillance network deployment requires it)
- the list of profiles supported

The process run-time options configuration file should define the following parameters:

```
addrFilter: 45.136.30-50.* #Montreal region
agentProf: disp
agentProf: .....
```

OR

```
addrFilter: 45.136.30.50/12 #Montreal region
agentProf: disp
agentProf: .....
```

Note: Default values are acceptable for the other parameters.

disp.agp file

The following example shows an agent profile (.agp) configuration file.

```
nodeType:GEN
deviceType: 51
oidFilter: 1.3.6.1.4.1.9    # all the Cisco traps
oidFilter: 1.3.6.1.2.1      # for generic traps
agentType: disp
```

This .agp configuration file contains the following specifications:

- The node type used is “GEN”; this does not matter because this profile never creates a component.
- The device type mainly used in the dcdAddNode script is 51.
- At least two object identifier (OID) filter elements must be defined:
  - The first is the OID of the MIB branch assigned to Cisco. Other filters should also be defined for other MIB branches shared by the different device types.
  - The second is the MIB-II OID used to match generic traps OID. More OID filter elements should be defined to handle the standard MIBs supported on Cisco devices.
- The agent type is “disp”. This is the name of the corresponding TYPE declaration in the .pol configuration file.

disp.pol file

The CLASS declaration of a polling (.pol) configuration file specifies

- device discovery
- device reachability

The TYPE declaration of a .pol configuration file specifies the polling requests associated with the disp agent type.

Device discovery

```
     CLASS dispDevDisc
     {
1    REQGROUP 0
2    COMPTYPE 0
3    METHOD single

4    sysOid  1.3.6.1.2.1.1.2.0

5    Next: LEFT(sysOid, "1.3.6.1.4.1.9.1")
6    Log: "MAJOR" SPCAT(sysOid, \
          "is not a Cisco product")
7    Discover: TRUE unsupported

8    Assign: prodNum TOKEN(sysOid, ".", 9, 1)
9    Next: NE(prodNum, 6)
10   Log: "MINOR" "Cisco3000 product identified"
11   Discover: TRUE cis3k

12   Next: NE(prodNum, 8)
13   Log: "MINOR" "Cisco7000 product identified"
14   Discover: TRUE cis7k

15   Next: AND(NE(prodNum, 173), NE(prodNum, 181), \
           NE(prodNum, 182))
16   Log: "MINOR" "Cisco 12000 product identified"
17   Discover: TRUE cis12k

18   Log: "MAJOR" SPCAT("Cisco product", sysOid, \
                        "is not supported")
19   Discover: TRUE unsupported
   }
```

Note: Some examples in this appendix contain numbers at the start of each line. These numbers are not part of the actual request; they have been added for clarity purposes to refer to individual lines.

This device discovery example is the polling request declaration contained in the configuration file disp.pol that is used to discover the device type.

Lines 1 to 3 define the request attributes:

- Line 1: the request declared is in the discovery group
- Line 2: the request is associated with the component type 0 (device)
- Line 3: an SNMP GET request is used

Line 4 is described as follows:

- Line 4: defines the polled variable (sysOid from RFC 1213)

Lines 5 to 19 specify commands executed to handle responses to this request:

- Line 5: verifies if the system OID is part of the ciscoProducts branch and jumps to the next block if it is
- Line 6: issues a “major” log because the device is not a Cisco product
- Line 7: uses the Discover command with an undefined profile. This causes the agent for this device to be put in the unmanaged state in which it is not polled, and the traps received are discarded.
- Line 8: extracts the OID element after ciscoProducts (1.3.6.1.4.1.9.1) from sysOid and places it in the prodNum variable
- Line 9: jumps to the next block if the product number is not 6
- Line 10: issues a “minor” log stating that the device is a Cisco 3000
- Line 11: executes the Discover command with the cis3K profile. This causes a new agent structure with this profile to be initialized. The current agent structure is destroyed.
- Line 12: jumps to the next block if the product number is not 8
- Line 13: issues a “minor” log stating that the device is a Cisco 7000

- Line 14: executes the Discover command with the cis7K profile. This causes a new agent structure with this profile to be initialized. The current agent structure is destroyed.
- Line 15: jumps to the next block if the product number is not 173, 181 or 182
- Line 16: issues a “minor” log stating that the device is part of the Cisco 12000 series
- Line 17: executes the Discover command with the cis12k profile. This causes a new agent structure with this profile to be initialized. The current agent structure is destroyed.
- Line 18: issues a “major” log stating that the device is not one of the Cisco products supported
- Line 19: executes the Discover command with a profile name that is not defined. This causes the agent for this device to be put in the unmanaged state in which it is not polled, and the traps it sends are discarded.

Reachability

```
      CLASS dispReachability
      {
1     REQGROUP 2
2     COMPTYPE 0
3     METHOD single

4     sysOid   1.3.6.1.2.1.1.2.0

5     Log: "INFO" SPCAT(DEVNAME(), "is reachable")
    }
```

This example is the polling declaration contained in the configuration file disp.pol that is used to verify the reachability of the device. The fact that a response is received already establishes the device reachability. In this example, the request is only used to issue an INFO log if this log level is selected.

Lines 1 to 3 define the request attributes:

- Line 1: the request declared is in the reachability group
- Line 2: the request is associated with the component type 0

- Line 3: an SNMP GET request is used

Lines 4 to 5 are described as follows:

- Line 4: defines the only polled variable; this is not used but is defined because a request must have at least one polled variable
- Line 5: issues the log if the level is selected

Agent type

The agent type declaration is included in the polling configuration file.

```
TYPE disp
{
  CLASSNAME dispDevDisc
  CLASSNAME dispReachability
}
```

This declaration lists all the polling request class names associated with the specified agent type.

The name of the agent type is the token following the TYPE keyword. This agent type name is associated with an agent profile in the .agp configuration file.

disp.tra file

The trap translation (.tra) file only contains a default translation record. This record specifies how to translate a trap into an alarm.

Default translation record

```
1   Trap_oid: DEFAULT
2   Trap_code: 0
3   Comp_id: E DEVNAME()
4   Fault_code: S C0009999
5   Severity: indeterminate
6   Event: message
7   Next: E NE(VERSION(), 1)
8   # This is an SNMPv2 trap
9   Comment: E SPCAT("SNMPv2 trap received; OID:", \
                     TRAPOID())

10  Trap_oid: DEFAULT
11  Trap_code: 0
12  Next: E EQ(GENERIC(), 6)
13  # This is a SNMPv1 generic trap
14  Comment: E CAT("SNMPv1 generic (", GENERIC(), ") \
                   trap received."

15  Trap_oid: DEFAULT
16  Trap_code: 0

17  #This is an enterprise SNMPv1 trap
18  Comment: E CAT("SNMPv1 enterprise trap received; \
                  OID:", TRAPOID(), \
                   "Specific code: ", SPECIFIC())
```

This translation record is only required if a trap is received from the device while this profile is activated, but before a response to the device discovery request has been received and processed.

- Lines 1, 2, 10, 11, 15, 16: identify the three translation blocks as default translations
- Lines 3 to 6: define the component id, fault code, severity, and event type for the three blocks
- Line 7: jumps to the next block if it is not an SNMPv2 trap

- Line 9: defines the comment attribute for an SNMPv2 trap
- Line 12: jumps to the next block if it is an enterprise trap
- Line 14: defines the comment attribute for an SNMPv1 generic trap
- Line 18: defines the comment attribute for an enterprise SNMPv1 trap

Preside Multiservice Data Manager

SNMP Surveillance Adapter

Guide

R15.1

Publication: 241-6001-118
Document status: Standard
Document version: 15.1RSUP
Document date: August 2004
Printed in Canada

NORTEL NETWORKS