MICROMUSE™
NETCOOL® SOLUTIONS

# Netcool®/OMNIbus™

v7

Probe and Gateway Guide

Micromuse Inc., 139 Townsend Street, San Francisco, USA CA 94107

www.micromuse.com

Document Version Number: 1.1

# Contents

# Preface

This guide describes how to configure and use probes and gateways.

It contains introductory and reference information about probes, including probe rules file syntax, properties and command line options, error messages, and troubleshooting techniques.

It also contains introductory and reference information about gateways, including gateway commands, command line options, and error messages.

For more information about specific probes and gateways, refer to the documentation available for each probe and gateway on the Micromuse Support Site.

This preface contains the following sections:

## Audience

This guide is intended for both users and administrators, and provides detailed cross-platform information about functions and capabilities. In addition, it is designed to be used as a reference guide to assist you in designing and configuring your environment.

Probes and gateways are part of Netcool/OMNIbus, and it is assumed that you understand how Netcool/OMNIbus works. For more information, refer to the publications described in *Associated Publications* on page 4.

# About the Netcool/OMNIbus v7 Probe and Gateway Guide

This book is organized as follows:

- Chapter 1: *Introduction to Probes* on page 9 introduces probes, their key features, and how to use them. It also describes the types of probes, their architecture and components, and how to run them.
- Chapter 2: *Probe Rules File Syntax* on page 27 describes rules file syntax. The rules file defines how the probe should process event data to create a meaningful Netcool/OMNIbus alert.
- Chapter 3: *Probe Properties and Command Line Options* on page 59 describes the properties and command line options common to all probes and TSMs.
- Chapter 4: *Introduction to Gateways* on page 69 introduces gateways, their key features, and how to use them. It also describes the types of gateways, their architecture and components, and how to run them.
- Chapter 5: *Gateway Commands and Command Line Options* on page 95 describes the command line options for `nco_gate`. It also describes gateway commands that are common to all gateways.
- Appendix A: *Regular Expressions* on page 119 contains information about how to use regular expressions.
- Appendix B: *ObjectServer Tables* on page 123 contains ObjectServer database table information. It describes the tables in the `alerts` and `service` databases and ObjectServer data types.
- Appendix C: *Probe Error Messages and Troubleshooting Techniques* on page 137 lists all of the messages that are common to all probes, including ProbeWatch and TSMWatch messages. It also includes troubleshooting information for probes.
- Appendix D: *Gateway Error Messages* on page 155 lists gateway error messages.

## Associated Publications

To use probes and gateways, you must possess an understanding of the Netcool/OMNIbus technology. This section provides a description of the documentation that accompanies Netcool/OMNIbus.

### Netcool®/OMNIbus™ Installation and Deployment Guide

This book is intended for Netcool administrators who need to install and deploy Netcool/OMNIbus. It includes installation, upgrade, and licensing procedures. In addition, it contains information about configuring security and component communications. It also includes examples of Netcool/OMNIbus architectures and how to implement them.

### Netcool®/OMNIbus™ User Guide

This book is intended for anyone who needs to use Netcool/OMNIbus desktop tools on UNIX or Windows platforms. It provides an overview of Netcool/OMNIbus components, as well as a description of the operator tasks related to event management using the desktop tools.

### Netcool®/OMNIbus™ Administration Guide

This book is intended for system administrators who need to manage Netcool/OMNIbus. It describes how to perform administrative tasks using the Netcool/OMNIbus Administrator GUI, command line tools, and process control. It also contains descriptions and examples of ObjectServer SQL syntax and automations.

### Netcool®/OMNIbus™ Probe and Gateway Guide

This guide contains introductory and reference information about probes and gateways, including probe rules file syntax and gateway commands. For more information about specific probes and gateways, refer to the documentation available for each probe and gateway on the Micromuse Support Site.

### Online Help

Netcool/OMNIbus GUIs contain context-sensitive online help with index and search capabilities.

# Typographical Notation

Table 1 shows the typographical notation and conventions used to describe commands, SQL syntax, and graphical user interface (GUI) features. This notation is used throughout this book and other Netcool® publications.

Table 1: Typographical Notation and Conventions (1 of 2)

| Example | Description |
|---|---|
| `Monospace` | The following are described in a monospace font:<br>• Commands and command line options<br>• Screen representations<br>• Source code<br>• Object names<br>• Program names<br>• SQL syntax elements<br>• File, path, and directory names<br>Italicized monospace text indicates a variable that the user must populate. For example, `-password` *`password`*. |
| **Bold** | The following application characteristics are described in a bold font style:<br>• Buttons<br>• Frames<br>• Text fields<br>• Menu entries<br>A bold arrow symbol indicates a menu entry selection. For example, **File→Save**. |
| *Italic* | The following are described in an italic font style:<br>• An application window name; for example, the *Login* window<br>• Information that the user must enter<br>• The introduction of a new term or definition<br>• Emphasized text |

Table 1: Typographical Notation and Conventions (2 of 2)

| Example | Description |
|---|---|
| [1] | Code or command examples are occasionally prefixed with a line number in square brackets. For example:<br>[1] `First command...`<br>[2] `Second command...`<br>[3] `Third command...` |
| `{ a \| b }` | In SQL syntax notation, curly brackets enclose two or more required alternative choices, separated by vertical bars. |
| `[ ]` | In SQL syntax notation, square brackets indicate an optional element or clause. Multiple elements or clauses are separated by vertical bars. |
| \| | In SQL syntax notation, vertical bars separate two or more alternative syntax elements. |
| `...` | In SQL syntax notation, ellipses indicate that the preceding element can be repeated. The repetition is unlimited unless otherwise indicated. |
| `,...` | In SQL syntax notation, ellipses preceded by a comma indicate that the preceding element can be repeated, with each repeated element separated from the last by a comma. The repetition is unlimited unless otherwise indicated. |
| <u>a</u> | In SQL syntax notation, an underlined element indicates a default option. |
| `( )` | In SQL syntax notation, parentheses appearing within the statement syntax are part of the syntax and should be typed as shown unless otherwise indicated. |

Many Netcool commands have one or more command line options that can be specified following a hyphen (`-`).

Command line options can be `string`, `integer`, or `BOOLEAN` types:

- A `string` can contain alphanumeric characters. If the string has spaces in it, enclose it in quotation (`"`) marks.
- An `integer` must contain a positive whole number or zero (`0`).
- A `BOOLEAN` must be set to `TRUE` or `FALSE`.

SQL keywords are not case-sensitive, and may appear in uppercase, lowercase, or mixed case. Names of ObjectServer objects and identifiers are case-sensitive.

## Note, Tip, and Warning Information

The following types of information boxes are used in the documentation:

**Note:** Note is used for extra information about the feature or operation that is being described. Essentially, this is for extra data that is important but not vital to the user.

**Tip:** Tip is used for additional information that might be useful for the user. For example, when describing an installation process, there might be a shortcut that could be used instead of following the standard installation instructions.

**Warning:** Warning is used for highlighting vital instructions, cautions, or critical information. Pay close attention to warnings, as they contain information that is vital to the successful use of our products.

## Syntax and Example Subheadings

The following types of constrained subheading are used in the documentation:

### Syntax

Syntax subheadings contain examples of ObjectServer SQL syntax commands and their usage. For example:

```
CREATE DATABASE database_name;
```

### Example

Example subheadings describe typical or generic scenarios, or samples of code. For example:

```
<body>
   <img src="ChartView?template=barchart&format=PNG
    &request=image&chart=quote&width=800&height=400" border="0" height="400"
    width="800" alt="Events by Severity"
   >
</body>
```

## Operating System Considerations

All command line formats and examples are for the standard UNIX shell. UNIX is case-sensitive. You must type commands in the case shown in the book.

Unless otherwise specified, command files are located in the `$OMNIHOME/bin` directory, where `$OMNIHOME` is the UNIX environment variable that contains the path to the Netcool/OMNIbus home directory.

On Microsoft Windows platforms, replace `$OMNIHOME` with `%OMNIHOME%` and the forward slash (/) with a backward slash (\).

# Chapter 1: Introduction to Probes

This chapter introduces probes, their key features, and how to use them. It also describes the types of probes, their architecture and components, and how to run them.

For descriptions of common properties and command line options, see Chapter 3: *Probe Properties and Command Line Options* on page 59.

For information about using probe rules file syntax to define how the probe should process event data, see *Rules File* on page 16 and Chapter 2: *Probe Rules File Syntax* on page 27.

For descriptions of probe error messages and troubleshooting hints, see Appendix C: *Probe Error Messages and Troubleshooting Techniques* on page 137.

For more information about specific probes, see *Using a Specific Probe* on page 22 and the individual guides available for each probe on the Micromuse Support Site.

This chapter contains the following sections:

## 1.1 Probe Overview

Probes connect to an event source, detect and acquire event data, and forward the data to the ObjectServer as alerts. Probes use the logic specified in a rules file to manipulate the event elements before converting them into fields of an alert in the ObjectServer `alerts.status` table.

Figure 1 shows how probes fit into the Netcool/OMNIbus architecture.

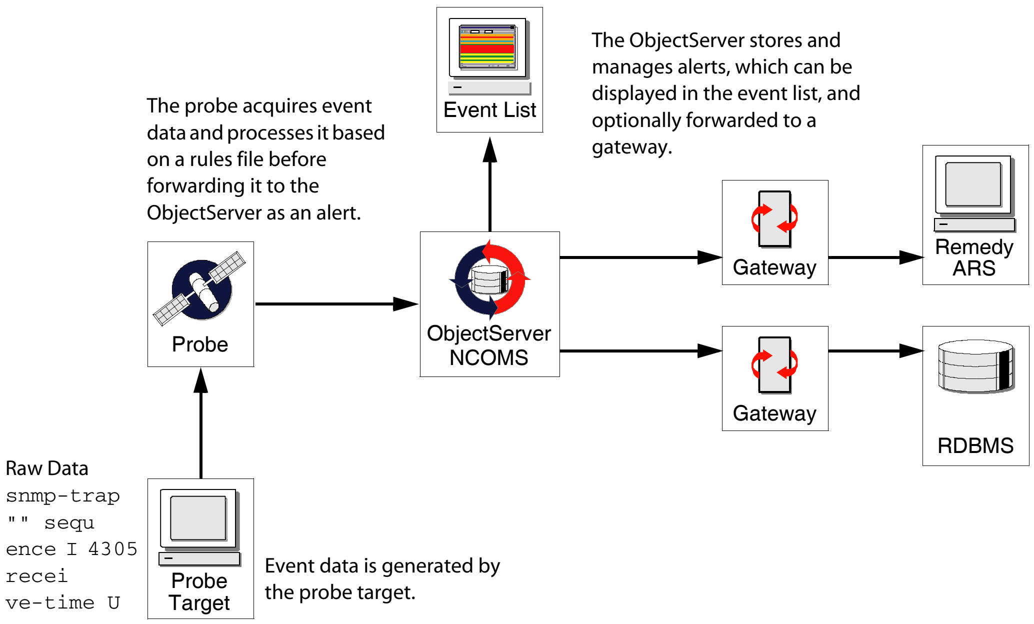

Event List
The ObjectServer stores and manages alerts, which can be displayed in the event list, and optionally forwarded to a gateway.
The probe acquires event data and processes it based on a rules file before forwarding it to the ObjectServer as an alert.
Probe
ObjectServer NCOMS
Gateway
Remedy ARS
Gateway
RDBMS
Raw Data
snmp-trap
" " sequ
ence I 4305
recei
ve-time U
Probe Target
Event data is generated by the probe target.

Figure 1: Event Processing in Netcool/OMNIbus

Probes can acquire data from any stable data source. These sources are described in *Types of Probes* on page 11.

## 1.2 Types of Probes

Each probe is uniquely designed to acquire event data from a specific source. However, probes can be categorized based on how they acquire events. For example, the Probe for Oracle obtains event data from a database table, and is therefore classed as a database probe. The types of probes are:

- Device
- Log file
- Database
- API
- CORBA
- Miscellaneous

These types of probes are described in the following sections.

**Tip:** The probe type is determined by the method in which the probe detects events. For example, the Probe for Agile ATM Switch Management detects events produced by a device (an ATM switch), but it acquires events from a log file, not directly from the switch. Therefore, this probe is classed as a log file probe and not a device probe.

### Device Probes

A device probe acquires events by connecting to a remote device, such as an ATM switch.

Device probes often run on a separate machine to the one they are probing and connect to the target machine through a network link, modem, or physical cable. Some device probes can use more than one method to connect to the target machine.

Once connected to the target machine, the probe detects events and forwards them to the ObjectServer. Some device probes are passive, waiting to detect an event before forwarding it to the ObjectServer; for example, the Probe for Marconi ServiceOn EMOS. Other device probes are more active, issuing commands to the target device in order to acquire events; for example, the TSM for Ericsson AXE10.

### Log File Probes

A log file probe acquires events by reading a log file created by the target system. For example, the Probe for Heroix RoboMon Element Manager reads the Heroix RoboMon Element Manager event file.

Most log file probes run on the machine where the log file resides; this is not necessarily the same machine as the target system. The target system appends events to the log file. Periodically, the probe opens the log file, acquires and processes the events stored in it, and forwards the relevant events to the ObjectServer as alerts. You can configure how often the probe checks the log file for new events and how events are processed.

## Database Probes

A database probe acquires events from a single database table; the *source* table. Depending on the configuration, any change (insert, update, or delete) to a row of the source table can produce an event. For example, the Probe for Oracle acquires data from transactions logged in an Oracle database table.

When a database probe is started, it creates a temporary logging table and adds a trigger to the source table. When a change is made to the source table, the trigger forwards the event to the logging table. Periodically, the events stored in the logging table are forwarded to the ObjectServer as alerts and the contents of the logging table are discarded. You can configure how often the probe checks the logging table for new events.

**Warning:** Existing triggers on the source table may be overwritten when the probe is installed.

Database probes treat each row of the source table as a single entity. Even if only one field of a row in the source table changes, all of the fields of that row are forwarded to the logging table and from there to the ObjectServer. If a row in the source table is deleted, the probe forwards the contents of the row before it was deleted. If a row in the source table is inserted or updated, the probe forwards the contents of the row after the insert or update.

## API Probes

An API probe acquires events through the API of another application. For example, the Probe for Sun Management Center uses the Sun Management Center Java API to connect remotely to the Sun Management Center.

API probes use specially designed libraries to acquire events from another application or management system. These libraries contain functions that connect to the target system and manage the retrieval of events. The API probes call these functions which connect to the target system and return any events to the probe. The probe processes these events and forwards them to the ObjectServer as alerts.

## CORBA Probes

Common Object Request Broker Architecture (CORBA) allows distributed systems to be defined independent of a specific programming language. CORBA probes use CORBA interfaces to connect to the data source; usually an Element Management System (EMS). Equipment vendors publish the details of their specific CORBA interface as Interface Definition Language (IDL) files. These IDL files are used to create the CORBA client and server applications. A specific probe is required for each specific CORBA interface.

CORBA probes use the Borland VisiBroker Object Request Broker (ORB) to communicate with other vendor's ORBs. You must obtain this ORB from Micromuse Support.

Most CORBA probes are written using Java, and require specific Java components to be installed to run the probe, as described in the individual guides for these probes. Probes written in Java use the following additional processes:

- The `nco_p_nonnative` probe, which enables probes written in Java to communicate with the standard probe C library (`libOpl`)
- Java runtime libraries

For example, the Probe for Marconi MV38/PSB manages the alarm lifecycle by collecting events from the Marconi ServiceOn Optical Network Management System. To do this, it connects to the Practical Service and Business (PSB) CORBA interface using the CORBA Naming Service running on the PSB host.

## Miscellaneous Probes

All of the miscellaneous probes have characteristics that differentiate them from the other types of probes and from each other. Each of them carries out a specialized task that requires them to work in a unique way.

For example, the Email Probe connects to the mail server, retrieves emails, processes them, deletes them, and then disconnects. This is useful on a workstation that does not have sufficient resources to permit an SMTP server and associated local mail delivery system to be kept resident and continuously running.

Another example of a probe in the miscellaneous category is the Ping Probe. It is used for general purpose applications on UNIX platforms and does not require any special hardware. You can use the Ping Probe to monitor any device that supports the ICMP protocol, such as switches, routers, PCs, and UNIX hosts.

## 1.3 Probe Components

A probe has the following primary components:

- An executable file
- A properties file
- A rules file

These components are described in the following sections.

**Tip:** Some probes have additional components. When additional components are provided, they are described in the individual probe guides.

### Executable File

The executable file is the core of a probe. It connects to the event source, acquires and processes events, and forwards the events to the ObjectServer as alerts.

Probe executable files are stored in the directory `$OMNIHOME/probes/`*`arch`*, where *`arch`* is the platform name of the architecture. For example, the executable file for the Ping Probe that runs on HP-UX 11.00 is:

```
$OMNIHOME/probes/hpux11/nco_p_ping
```

To start a probe on UNIX with the appropriate configuration information, run the wrapper script in the directory `$OMNIHOME/probes`. For example, to start the Ping Probe, enter:

```
$OMNIHOME/probes/nco_p_ping
```

When the probe is started, it obtains information on how to configure its environment from the properties and rules files, described in the next sections. The probe uses this configuration information to customize the data it forwards to the ObjectServer.

For more information about how to run a specific probe and specify command line options, see *Using a Specific Probe* on page 22.

### Properties File

Probe properties define the environment in which the probe runs. For example, the `Server` property specifies the ObjectServer to which the probe forwards alerts. Probe properties are stored in a properties file in the directory `$OMNIHOME/probes/`*`arch`*. Properties files are identified by the `.props` file extension.

For example, the properties file for the Ping Probe that runs on HP-UX 11.00 is:

```
$OMNIHOME/probes/hpux11/ping.props
```

Properties files are formed of name-value pairs separated by a colon. For example:

```
Server : "NCOMS"
```

In this name-value pair, `Server` is the name of the property and `NCOMS` is the value to which the property is set. String values must be enclosed in quotes; other values do not require quotes.

## Probe Property Types

Properties can be divided into two categories: common properties and probe-specific properties.

For example, the `Server` property is a common property, because every probe needs to know which ObjectServer to send alerts to. Common properties are described in *Probe Properties and Command Line Options* on page 60.

Probe-specific properties vary by probe. Some probes do not have any specific properties, but most have additional properties that relate to the environment in which they run. For example, the Ping Probe has a `Pingfile` property which specifies the name of a file containing a list of the machines to be pinged. Probe-specific properties are described in the individual probe guides, available on the Micromuse Support Site.

## Properties and Command Line Options

There is a command line option that corresponds to each probe property. For example, the `Server` property is set in the properties file:

```
Server : "NCOMS"
```

It can also be set on the command line using the option:

```
$OMNIHOME/probes/nco_p_probename -server STWO
```

The command line option overrides the property when both are set. In the preceding example, where the property sets the server to `NCOMS` and the command line option sets the server to `STWO`, the value `STWO` is used for the ObjectServer name. For more information on using command line options to override properties, refer to *Probe Properties and Command Line Options* on page 60.

## Rules File

The rules file defines how the probe should process event data to create a meaningful alert. The rules file also creates an identifier for each alert, the `Identifier` field, described in *Creating a Unique Identifier* on page 18. This identifier is used to uniquely identify the problem source. Duplicate alerts (those with the same identifier) are correlated so they only appear in the event list once.

Local rules files are stored in the directory `$OMNIHOME/probes/`*`arch`*, and are identified by the `.rules` file extension.

For example, the rules file for the Ping Probe that runs on HP-UX 11.00 is:

```
$OMNIHOME/probes/hpux11/ping.rules
```

You can use a URL to specify a rules file located on a remote server that is accessible using the `http` protocol. This allows all rules files to be sourced for each probe from a central point. Using a suitable configuration management tool, such as CVS, at the central point enables version management of all rules files.

Refer to Chapter 2: *Probe Rules File Syntax* on page 27 for detailed information about rules files and how to modify them.

### Re-reading the Rules File

For changes to the rules file to take effect, the probe must be forced to re-read the rules file. You can force the probe to re-read the rules file by issuing the command `kill -HUP` *`pid`* on the probe process ID (PID). Refer to the `ps` and `kill` man pages for more information.

This method is preferable to restarting the probe, because the probe will not lose events.

**Tip:** For CORBA probes, issue the command `kill -HUP` on the `nco_p_nonnative` process.

## 1.4 Probe Architecture

The function of a probe is to acquire information from an event source and forward it to the ObjectServer. Figure 2 shows how probes process the event data acquired from the event source using rules.

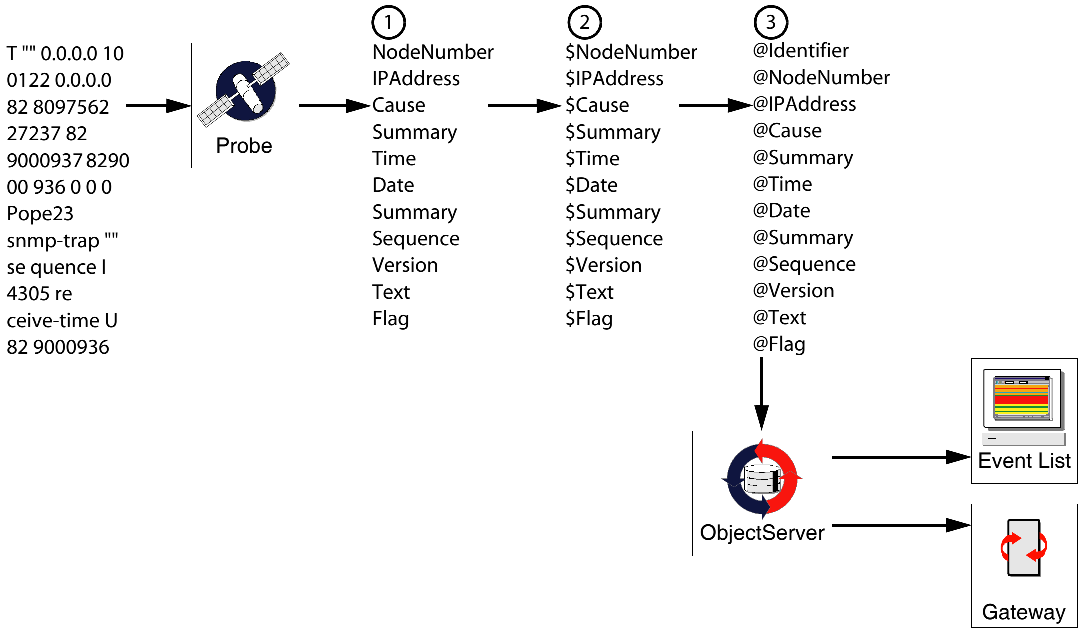


Figure 2: Event Mapping Using Rules

The raw event data that a probe acquires cannot be sent directly to the ObjectServer. The probe breaks the event data into tokens (1 - in Figure 2). Each token represents a piece of event data.

The probe then parses these tokens into elements and processes the elements according to the rules in the rules file (2 - in Figure 2). Elements are identified in the rules file by the `$` symbol. For example, `$Node` is an element containing the node name of the event source.

Elements are used to assign values to ObjectServer fields, indicated by the `@` symbol (3 - in Figure 2). The field values contain the event details in a form understood by the ObjectServer. Fields make up the alerts which are forwarded to the ObjectServer, where they are stored and managed in the `alerts.status` table and displayed in the event list.

The `Identifier` field is also generated by the rules file, as described in the next section.

For more information about manipulating fields and elements, see Chapter 2: *Probe Rules File Syntax* on page 27.

## 1.5 Creating a Unique Identifier

The `Identifier` field (`@Identifier`) is used to uniquely identify a problem source. Like other ObjectServer fields, the `Identifier` field is constructed from the tokens the probe acquires from the event stream according to the rules in the rules file.

The `Identifier` field allows the ObjectServer to correlate alerts so that duplicate alerts only appear in the event list once. Rather than inserting a new alert, the alert is *reinserted* —the existing alert is updated. These updates are configurable. For example, the tally field (`@Tally`) is typically incremented to keep track of the number of times the event occurred.

It is essential that the identifier identifies repeated events appropriately. The following identifier is not specific enough, because any events with the same manager and node are treated as duplicates:

```
@Identifier=@Manager+@Node
```

If the identifier is too specific, the ObjectServer is not able to correlate and deduplicate repeated events. For example, an identifier that contains a time value prevents correct deduplication.

The following identifier correctly identifies repeated events in a typical environment:

```
@Identifier=@Node+" "+@AlertKey+" "+@AlertGroup+" "+@Type+" "+@Agent+" "+@Manager
```

Rules file syntax is described in Chapter 2: *Probe Rules File Syntax* on page 27.

## Deduplication with Probes

Deduplication is managed by the ObjectServer, but can be configured in the probe rules file. This enables you to set deduplication rules on a per-event basis. You can specify which fields of an alert are to be updated if the alert is deduplicated using the `update` function. This is described in *Update on Deduplication Function* on page 47.

For an overview of alert processing and deduplication, see the Netcool/OMNIbus Administration Guide.

## 1.6 Probe Features

This section describes some of the key features of probe operation.

## Store and Forward Mode

Probes can continue to run if the target ObjectServer is down. When the probe detects that the ObjectServer is not present (usually because it is unable to forward an alert to the ObjectServer), it switches to *store* mode. In this mode, the probe writes all of the messages it would normally send to the ObjectServer to a file named:

```
$OMNIHOME/var/probename.servername.store
```

In this file name, `probename` is the name of the probe and `servername` is the name of the ObjectServer to which the probe is attempting to send alerts.

When the probe detects that the ObjectServer is back on line, it switches to *forward* mode and sends the alert information held in the `.store` file to the ObjectServer. Once all of the alerts in the `.store` file have been forwarded, the probe returns to normal operation.

Store and forward functionality is enabled by default, but can be disabled by setting the `StoreAndForward` property to `0` (`FALSE`) in the properties file or using the `-nosaf` command line option.

The `RetryConnectionCount`, `RetryConnectionTimeout`, and `MaxSAFFileSize` properties also control the operation of store and forward mode. Refer to *Probe Properties and Command Line Options* on page 60 for more information about these properties.

### Automatic Store and Forward

By default, store and forward mode is active only after a connection to the ObjectServer has been established, used, and then lost. If the ObjectServer is not running when the probe starts, store and forward mode is not triggered and the probe terminates.

However, if you set the probe to run in *automatic* store and forward mode, it will go straight into store mode if the ObjectServer is not running, as long as the probe has been connected to the ObjectServer at least once before. Enable automatic store and forward mode using the `-autosaf` command line option or `AutoSAF` property.

**Note:** For failover to work, automatic store and forward must be enabled in addition to setting the `ServerBackup`, `NetworkTimeout`, and `PollServer` properties.

## Raw Capture Mode

Raw capture mode enables you to save the complete stream of event data acquired by a probe into a file without any processing by the rules file. This can be useful for auditing, recording, or debugging the operation of a probe.

The captured data is in a format that can be replayed by the Generic Probe, as described in the guide for the Generic Probe.

You can enable raw capture mode using the `-raw` command line option or `RawCapture` property.

You can also set the `RawCapture` property in the rules file, so that you can send the raw event data to a file only when certain conditions are met. See *Changing the Value of the RawCapture Property in the Rules File* on page 30 for more information.

The `RawCaptureFile`, `RawCaptureFileAppend`, and `MaxRawFileSize` properties also control the operation of raw capture mode. Refer to *Probe Properties and Command Line Options* on page 60 for more information about these properties.

## Secure Mode

You can run the ObjectServer in secure mode. When you start the ObjectServer using the `-secure` command line option, the ObjectServer authenticates probe, gateway, and proxy server connections by requiring a user name and encrypted password. When a connection request is sent, the ObjectServer issues an authentication message. The probe, gateway, or proxy server must respond with the correct user name and password combination.

If the ObjectServer is not running in secure mode, probe, gateway, and proxy server connection requests are not authenticated.

When connecting to a secure ObjectServer or proxy server, each probe must have the `AuthUserName` and `AuthPassword` properties in its property file. If the user name and password combination is incorrect, the ObjectServer issues an error message and rejects the connection.

You must encrypt the passwords used in secure mode using the `nco_g_crypt` utility, described in the Netcool/OMNIbus Administration Guide. Then, add the `AuthUserName` and `AuthPassword` properties to the probe properties file with the corresponding user name and encrypted password before running the probe.

## Peer-to-Peer Failover

Two instances of a probe can run simultaneously in a peer-to-peer failover relationship. One instance is designated as the master; the other acts as a slave and is on hot standby. If the master instance fails, the slave instance is activated.

**Note:** Peer-to-peer failover is not supported for all probes. Probes that list the `Mode`, `PeerHost`, and `PeerPort` properties when you run the command `$OMNIHOME/probes/nco_p_`*`probename`* `-dumpprops` support peer-to-peer failover.

To set up a peer-to-peer failover relationship, do the following:

- For the master instance, set the `Mode` property to `master` and the `PeerHost` property to the network element name of the slave.
- For the slave instance, set the `Mode` property to `slave` and the `PeerHost` property to the network element name of the master.
- For both instances, set the `PeerPort` property to the port through which the master and slave communicate.

To disable the peer-to-peer failover relationship, run a single instance of the probe with the `Mode` property set to `standard`. This is the default setting.

## Setting the Failover Mode in the Properties Files

The failover mode is set in the properties files. The following are example properties file values for the master:

```
PeerPort: 9999
PeerHost: "slavehost"
Mode: "master"
```

The following are example properties file values for the slave:

```
PeerPort: 9999
PeerHost: "masterhost"
Mode: "slave"
```

## Setting the Mode in the Rules File

The mode of a probe can be switched between master and slave in the rules file. For example, to switch a probe instance to become the master, use the rules file syntax:

```
%Mode = "master"
```

There is a delay of up to one second before the mode change takes effect. This can result in duplicate events if two probe instances are switching from `standard` mode to `master` or `slave`; however, no data is lost.

## 1.7 Using a Specific Probe

Each probe has an abbreviated name that is used to identify the probe executable and its associated files. For example, the abbreviated name for SunNet Manager is `snmlog` and the abbreviated name for IBM Netview/6000 is `nv`.

The Probe for SunNet Manager executable is named:

```
$OMNIHOME/probes/arch/nco_p_snmlog
```

**Tip:** To start the probe on UNIX with the appropriate configuration, run the wrapper script, as described in *Running a Probe on UNIX* on page 22.

The properties file is named:

```
$OMNIHOME/probes/arch/snmlog.prop
```

The rules file is named:

```
$OMNIHOME/probes/arch/snmlog.rules
```

In these paths, *`arch`* is the name of the architecture on which the probe is installed; for example, `solaris2` when running on a Solaris system.

Refer to the guide for each probe, available on the Micromuse Support Site, for details about specific probes, their defaults, and which of their properties can be changed.

## Running a Probe on UNIX

This section describes how to run a probe from the command line.

**Note:** Probes should be managed by process control. Process control is described in the Netcool/OMNIbus Administration Guide.

Once you have installed the probe, you must configure the properties and rules files to fit your environment. For example, if you are using a log file probe such as the HTTP Common Log Format Probe, you need to set the `LogFile` property, so that the probe can connect to the event source. Refer to *Probe Components* on page 14 for more information about properties and rules files.

To run a probe, enter:

```
$OMNIHOME/probes/nco_p_probename [ -option [ value ] ... ]
```

The `probename` is the abbreviated name of the probe you want to run. The `-option` is the command line option and `value` is the value you are setting the option to. Not every option requires you to specify a value. For example, to run the Sybase Probe in raw capture mode, enter:

```
$OMNIHOME/probes/nco_p_sybase -raw
```

The command line options available to all probes are described in Chapter 3: *Probe Properties and Command Line Options* on page 59.

**Note:** If you are running a proxy server, connect your probes to the proxy server rather than to the ObjectServer. To do this, use the `Server` property or `-server` command line option and specify the name of the proxy server. For more information on the proxy server, see the Netcool/OMNIbus Administration Guide.

## Running a Probe on Windows

You can run probes on Windows as console applications or services.

**Tip:** In versions of Netcool/OMNIbus prior to v7, probes were installed as services by default. As of Netcool/OMNIbus v7, probes are installed as console applications by default.

### Running a Probe as a Console Application

To run a probe as a console application, enter the following command from the probe directory:

```
nco_p_probename [ -option [ value ] ... ]
```

In this command, `probename` is the abbreviated name of the probe you want to run and `option` is a command line option.

There are extra command line options available for the Windows version of each probe. To display these, enter the following command:

```
nco_p_probename -?
```

The Windows-specific command line options are described in Table 2.

Table 2: Windows-Specific Probe Command Line Options

| Command Line Option | Description |
|---|---|
| `-install` | This option installs the probe as a Windows service. |
| `-noauto` | This option is used with the `-install` option. It disables automatic startup for the probe running as a service. If this option is used, the probe is not started automatically when the machine boots. |
| `-remove` | This option removes a probe that is installed as a service. It is the opposite of the `-install` command. |
| `-group` *`string`* | This option is used with the `-depend` command line option. You can group all the probes together under the same group name. You can then force that group to be dependent on another service. |
| `-depend` *`srv @grp`* `...` | This option specifies other services or groups that the probe is dependent on. If you use this option, the probe will not start until the services (*`srv`*) and groups (`@`*`grp`*) specified with this option have been run. |
| `-cmdLine "`*`-option value`*`..."` | This option specifies one or more command line options to be set each time the probe service is restarted. |

## Running a Probe as a Service

To run a probe as a service, use the `-install` command line option.

Configure how probes are started using the *Services* window as follows:

1. Click **Start**→**Settings**→**Control Panel**. The Control Panel is displayed.
2. Double-click the **Admin Tools** icon, then double-click the **Services** icon. The *Services* window is displayed.

   The *Services* window lists all of the Windows services currently installed on your machine. All Netcool/OMNIbus service names start with `NCO`.

3. Use the *Services* window to start and stop Windows services. Indicate whether the service is started automatically when the machine is booted by clicking the **Startup** button.

**Note:** If the ObjectServer and the probe are started as services, the probe may start first. The probe will not be able to connect to the ObjectServer until the ObjectServer is running.

# Chapter 2: Probe Rules File Syntax

This chapter describes rules file syntax. The rules file defines how the probe should process event data to create a meaningful Netcool/OMNIbus alert. The rules file also creates an identifier for each alert to uniquely identify the problem source, so repeated events can be deduplicated.

For introductory information about rules files, see *Rules File* on page 16. For information about the `Identifier` field, see *Creating a Unique Identifier* on page 18.

This chapter contains the following sections:

## 2.1 Elements, Fields, Properties, and Arrays in Rules Files

A probe takes an event stream and parses it into elements. Event elements are processed by the probe based on the logic in the rules file. Elements are assigned to fields and forwarded to the ObjectServer, where they are inserted as alerts into the `alerts.status` table.

The `Identifier` field, used by the ObjectServer for deduplication, is also created based on the logic in the rules file, as described in *Creating a Unique Identifier* on page 18.

Elements are indicated by the `$` symbol in the rules file. For example, `$Node` is an element containing the node name of the event source. You can assign elements to ObjectServer fields, indicated by the `@` symbol in the rules file.

### Assigning Values to ObjectServer Fields

You can assign values to ObjectServer fields in the following ways:

- Direct assignment, for example: `@Node = $Node`
- Concatenation, for example: `@Summary = $Summary + $Group`
- Adding text, for example: `@Summary = $Node + "has problem" + $Summary`

Numeric values can be expressed in decimal or hexadecimal form. The following statements, which set the `Class` field to `100`, are equivalent:

- `@Class=100`
- `@Class=0x64`

In addition to assigning elements to fields, you can use the processing statements, operators, and functions described in this chapter to manipulate these values in rules files before assigning them.

**Tip:** Elements are stored as strings, so you need to use the `int` function, described in *Math Functions* on page 43, to convert them into integers before performing numeric operations.

### Assigning Temporary Elements in Rules Files

You can create a temporary element in a rules file by assigning it to an expression, for example:

```
$tempelement = "message"
```

An element, `$tempelement`, is created and assigned the string value `message`.

If you refer to an element that has not been initialized in this way, the element is set to the null string (`""`).

The following example creates the element `$b` and sets it to `setnow`:

```
$b="setnow"
```

The following example then sets the element `$a` to `setnow`:

```
$a=$b
```

In the following example, temporary elements are used to extract information from a `Summary` element, which has the string value: `The Port is down on Port 1 Board 2`.

```
$temp1 = extract ($Summary, "Port ([0-9]+)")
$temp2 = extract ($Summary, "Board ([0-9]+)")
@AlertKey = $temp1 + "." + $temp2
```

The `extract` function is used to assign values to temporary elements `temp1` and `temp2`. Then these elements are concatenated with a `.` separating them, and assigned to the `AlertKey` field. After these statements are executed, the `AlertKey` field has the value `1.2`. The `extract` function is described in *String Functions* on page 41, and the concatenate operator (+) is described in *Math and String Operators* on page 37.

## Assigning Property Values to Fields

You can assign the value of a probe property, as defined in the properties file or on the command line, to a field value. A property is represented by a `%` symbol in the rules file. For example, you could add the following statement to your rules file:

```
@Summary = "Server = " + %Server
```

In this example, when the rules file is processed, the probe searches for a property named `Server`. If the property is found, its value is concatenated to the text string and assigned to the `Summary` field. If the property is not found, a null string (`""`) is assigned.

## Assigning Values to Properties

You can also assign values to a property in the rules file. If the property does not exist, it is created. For example, you could create a property called `Counter` to keep track of the number of events that have been processed as follows:

```
if (match(%Counter,""))
    {%Counter = 1}
else {%Counter = int(%Counter) + 1}
```

These properties retain their values across events and when the rules file is re-read.

### Changing the Value of the RawCapture Property in the Rules File

Most probes read properties once at startup, so changing probe properties after startup does not usually affect probe behavior. However, the `RawCapture` property can be set in the rules file, so that you can send the raw event data to a file only when certain conditions are met. For example:

```
# Start rules processing
%RawCapture=0

if (condition) {
     # Send the next event to the raw capture file
     %RawCapture=1
}
```

The `IF` statement is described in *Conditional Statements in Rules Files* on page 32.

**Tip:** The setting for raw capture mode takes effect for the next event processed; not for the current event.

You can enable raw capture mode globally by setting the `-raw` command line option or the `RawCapture` property in the probe properties file, as described in *Raw Capture Mode* on page 20.

## Using Arrays

You must define arrays at the start of a rules file, before any processing statements.

**Tip:** You must also define tables, described in *Lookup Table Operations* on page 45, and target ObjectServers, described in *Sending Alerts to Alternate ObjectServers and Tables* on page 50, before any processing statements.

To define an array, use the following syntax:

```
array node_arr;
```

Arrays are one dimensional. Each time an assignment is made for a key value that already exists, the previous value is overwritten. For example:

```
node_arr["myhost"] = "a";
node_arr["yourhost"] = "b";
node_arr["myhost"] = "c";
```

After the preceding statements have been executed, there are two items in the `node_arr` array. The item with the key `myhost` is set to `c`, and the item with the key `yourhost` is set to `b`. You can make assignments using probe elements, for example:

```
node_arr[$Node] = "d";
```

**Note:** Array values are persistent until a probe is restarted; if you force the probe to re-read the rules file by issuing a `kill -HUP pid` command on the probe process ID, the array values are maintained.

## 2.2 Conditional Statements in Rules Files

The IF and SWITCH statements provide condition testing for processing elements in rules files.

### The IF Statement

A condition is a combination of expressions and operations that resolve to either TRUE or FALSE. The IF statement allows conditional execution of a set of one or more assignment statements by executing only the rules for the condition that is TRUE. It has the following syntax:

```
if (condition) {
     rules
}    [ else if (condition) {
     rules
}    ... ]
     [ else (condition) {
     rules
} ]
```

You can combine conditions into increasingly complex conditions using the logical AND operator (&&), which is true only if *all* of its inputs are true, and OR operator (||), which is true if *any* of its inputs are true. For example:

```
if match ($Enterprise, "Acme") && match ($trap-type, "Link-Up") {
@Summary = "Acme Link Up on " + @Node
}
```

Logical operators are described in *Logical Operators* on page 39. The match function is described in *String Functions* on page 41.

### The SWITCH Statement

A SWITCH statement transfers control to a set of one or more rules assignment statements depending on the value of an expression. It has the following syntax:

```
switch (expression) {
case "stringliteral":
     rules
case "stringliteral":
     rules
...
default:
     [rules]
}
```

The SWITCH statement tests for exact matches only. This statement should be used wherever possible instead of an IF statement because SWITCH statements are processed more efficiently and therefore execute more quickly.

**Note:** Each SWITCH statement must contain a default case even if there are no rules associated with it. There is no BREAK clause for the SWITCH statement, so any rules in the DEFAULT case are executed if no other case is matched.

The *expression* can be any valid expression. For example:

```
switch($node)
```

The *stringliteral* can be any string value. For example:

```
case "jupiter":
```

You can have more than one *stringliteral* separated by the pipe (|) symbol. For example:

```
case "jupiter" | "mars" | "venus":
```

This case is executed if the expression matches any of the specified strings.

## 2.3 Including Multiple Rules Files

You can include a number of secondary rules files in your main rules file using the `include` statement:

```
include "rulesfile"
```

Specify the path to the rules file as an absolute path. A relative path is relative to the probe working directory, which can vary depending on how the probe is started. You cannot use environment variables in the path.

```
if(match(@Manager, "ProbeWatch"))
{
include "/opt/netcool/omnibus/probes/solaris2/probewatch.rules"
}
else
...
```

## 2.4 Rules File Functions and Operators

You can use the operators and functions described in this section to manipulate elements in rules files before assigning them to ObjectServer fields.

Table 3 lists the operators described in the following sections.

Table 3: Rules File Operators

| Operators | Description | Details on... |
|---|---|---|
| `*`, `/`, `-`, `+` | Perform math and string operations. | page 37 |
| `&`, `\|`, `^`, `>>`, `<<` | Perform bitwise operations. | page 38 |
| `==`, `!=`, `<>`, `<`, `>`, `<=`, `>=` | Perform comparison operations. | page 38 |
| `NOT` (also `!`), `AND` (also `&&`), `OR` (also `\|\|`), `XOR` (also `^`) | Perform logical (boolean) operations. | page 39 |

Table 4 lists the functions described in the following sections.

Table 4: Rules File Functions (1 of 3)

| Function Name | Description | Details on... |
|---|---|---|
| `datetotime` | Converts a string into a time data type. | page 44 |
| `details` | Adds information to the `alerts.details` table. | page 48 |
| `discard` | Deletes an entire event. | page 40 |
| `exists` | Tests for the existence of an element. | page 39 |
| `expand` | Returns a string (which must be a literal string) with escape sequences expanded. | page 41 |
| `extract` | Returns the part of a string (which can be a field, element, or string expression) that matches the parenthesized section of the regular expression. | page 41 |
| `getdate` | Returns the current date as a date data type. | page 44 |
| `getenv` | Returns the value of an environment variable. | page 45 |
| `getload` | Measures the load on the ObjectServer. | page 52 |
| `getpid` | Returns the process ID of a running probe. | page 45 |
| `hostname` | Returns the name of the host on which the probe is running. | page 45 |

Table 4: Rules File Functions (2 of 3)

| Function Name | Description | Details on... |
|---|---|---|
| `int` | Converts a numeric value into an integer. | page 43 |
| `length` | Calculates the length of an expression and returns the numeric value. | page 41 |
| `log` | Enables you to log messages. | page 49 |
| `lookup` | Uses a lookup table to map additional information to an alert. | page 45 |
| `lower` | Converts an expression to lowercase. | page 41 |
| `ltrim` | Removes white space from the left of an expression. | page 41 |
| `match` | Tests for an exact string match. | page 41 |
| `nmatch` | Tests for a string match at the beginning of a specified string. | page 41 |
| `printable` | Converts any non-printable characters in an expression to a space character. | page 41 |
| `real` | Converts a numeric value into a real number. | page 43 |
| `recover` | Recovers a discarded event. | page 40 |
| `regmatch` | Performs full regular expression matching of a value in a regular expression in a string. | page 41 |
| `remove` | Removes an element from an event. | page 40 |
| `registertarget` | Registers an ObjectServer so alerts can be sent to multiple ObjectServers. | page 50 |
| `rtrim` | Removes white space from the right of an expression. | page 41 |
| `scanformat` | Converts an expression according to the available formats, similar to the `scanf` family of routines in C. | page 41 |
| `setlog` | Enables you to set the message log level. | page 49 |
| `settarget`, `setdefaulttarget` | Sets the ObjectServer alerts are sent to. | page 50 |
| `service` | Sets the status of a service. | page 51 |
| `split` | Separates a string into elements of an array. | page 41 |
| `substr` | Extracts a substring from an expression. | page 41 |

Table 4: Rules File Functions (3 of 3)

| Function Name | Description | Details on... |
|---|---|---|
| `timetodate` | Converts a time value into a string data type. | page 44 |
| `update` | Indicates which fields are updated when an alert is deduplicated. | page 47 |
| `updateload` | Updates the load statistics for the ObjectServer. | page 52 |
| `upper` | Converts an expression to uppercase. | page 41 |

## Math and String Operators

You can use math operators to add, subtract, divide, and multiply numeric operands in expressions. Table 5 describes the math operators supported in rules files.

Table 5: Math Operators

| Operator | Description | Example |
|---|---|---|
| *<br>/ | Operators used to multiply (`*`) or divide (`/`) two operands. | `$eventid=int($eventid)*2` |
| +<br>- | Operators used to add (+) or subtract (−) two operands. | `$eventid=int($eventid)+1` |

You can use string operators to manipulate character strings. Table 6 describes the string operator supported in rules files.

Table 6: String Operator

| Operator | Description | Example |
|---|---|---|
| + | Concatenates two or more strings. | `@field = $element1 + "message" + $element2` |

## Bit Manipulation Operators

You can use bitwise operators to manipulate integer operands in expressions. Table 7 describes the bitwise operators supported in rules files.

Table 7: Bitwise Operators

| Operator | Description | Example |
|---|---|---|
| `&`<br>`\|`<br>`^` | Bitwise AND (&), OR (\|), and XOR (^). The results are determined bit-by-bit. | `$result1 = int($number1) & int($number2)` |
| `>>`<br>`<<` | Shifts bits right (>>) or left (<<). | `$result2 = int($number3) >> 1` |

These operators manipulate the bits in integer expressions. For example, in the statement:

```
$result2 = int($number3) >> 1
```

If `number3` has the value `17`, `result2` resolves to `8`, as shown:

```
      16  8  4  2  1
>>     1  0  0  0  1
       0  1  0  0  0
```

Note that the bits do not wrap around; when they drop off one end, they are replaced on the other end by a `0`.

Bitwise operators only work with integer expressions. Elements are stored as strings, so you need to use the `int` function, described in *Math Functions* on page 43, to convert them into integers before performing these operations.

## Comparison Operators

You can use comparison operators to test numeric values for equality and inequality. Table 8 describes the comparison operators supported in rules files.

Table 8: Comparison Operators (1 of 2)

| Operator | Description | Example |
|---|---|---|
| `==` | Tests for equality. | `int($eventid) == 5` |
| `!=`<br>`<>` | Tests for inequality. | `int($eventid) != 0` |

Table 8: Comparison Operators (2 of 2)

| Operator | Description | Example |
|---|---|---|
| `<`<br>`>`<br>`<=`<br>`>=` | Tests for greater than (>), less than (<), greater than or equal to (>=), or less than or equal to (<=). | `int($eventid) <=30` |

## Logical Operators

You can use logical operators on boolean values to form expressions that resolve to `TRUE` or `FALSE`. Table 9 describes the logical operators supported in rules files.

Table 9: Logical Operators

| Operator | Description | Example |
|---|---|---|
| `NOT` (also `!`) | A `NOT` expression negates the input value, and is `TRUE` only if its input is `FALSE`. | `if NOT(Severity=0)` |
| `AND` (also `&&`) | An `AND` expression is true only if all of its inputs are `TRUE`. | `if match($Enterprise,"Acme") &&`<br>`match($trap-type,"Link-Up")` |
| `OR` (also `\|\|`) | An `OR` expression is `TRUE` if any of its inputs are `TRUE`. | `if match($Enterprise,"Acme") \|\|`<br>`match($Enterprise,"Bo")` |
| `XOR` (also `^`) | An `XOR` expression is `TRUE` if either of its inputs, but not both, are `TRUE`. | `if match($Enterprise,"Acme") XOR`<br>`match($Enterprise,"Bo")` |

## Existence Function

You can use the `exists` function to test for the existence of an element, with the following syntax:

```
exists ($element)
```

The function returns `TRUE` if the element was created for this particular event; otherwise it returns `FALSE`.

## Deleting Elements or Events

You can use functions to remove elements from an event, discard an entire event, and recover a discarded event. Table 10 describes these functions:

Table 10: Deleting Elements or Events

| Function | Description | Example |
|---|---|---|
| `discard` | Deletes an entire event.<br>*This must be in a conditional statement; otherwise, all events are discarded.* | `if match(@Node,"testnode") { discard }` |
| `recover` | Recovers a discarded event. | `if match(@Node,"testnode") { recover }` |
| `remove(`*`element_name`*`)` | Removes the element from the event. | `remove(test_element)` |

## String Functions

You can use string functions to manipulate string elements, typically field or element names. Table 11 describes the string functions supported in rules files.

Table 11: String Functions (1 of 3)

| Function | Description | Example |
|---|---|---|
| `expand("`*`string`*`")` | Returns the string (which must be a literal string) with escape sequences expanded. Possible expansions are:<br>`\"` - double quote<br>`\NNN` - octal value of `NNN`<br>`\\` - backslash<br>`\a` - alert (BEL)<br>`\b` - backspace<br>`\e` - escape (033 octal)<br>`\f` - form feed<br>`\n` - new line<br>`\r` - carriage return<br>`\t` - horizontal tab<br>`\v` - vertical tab<br>This function cannot be used as the regular expression argument in the `regmatch` or `extract` functions. | `log(debug, expand("Rules file with embedded \\\""))`<br>sends the following to the log:<br>`Sun Oct 21 19:56:15 2001 Debug: Rules file with embedded \"` |
| `extract(`*`string`*`, "`*`regexp`*`")` | Returns the part of the string (which can be a field, element, or string expression) that matches the parenthesized section of the regular expression. Regular expression pattern matching is described in Appendix A: *Regular Expressions* on page 119. | `extract ($expr,"ab([0-9]+)cd")`<br>If `$expr` is "`ab123cd`" then the value returned is `123`. |
| `length(`*`expression`*`)` | Calculates the length of an expression and returns the numeric value. | `$NodeLength = length($Node)` |
| `lower(`*`expression`*`)` | Converts an expression to lowercase. | `$Node = lower($Node)` |

Table 11: String Functions (2 of 3)

| Function | Description | Example |
|---|---|---|
| `ltrim(`*`expression`*`)` | Removes white space from the left of an expression. | `$TrimNode = ltrim($Node)` |
| `match(`*`expression`*`, "`*`string`*`")` | `TRUE` if the expression value matches the string exactly. | `if match($Node, "New")` |
| `nmatch(`*`expression`*`, "`*`string`*`")` | `TRUE` if the expression starts with the specified string. | `if nmatch($Node, "New")` |
| `printable(`*`expression`*`)` | Converts any non-printable characters in the given expression into a space character. | `$Print = printable($Node)` |
| `regmatch(`*`expression`*`, "`*`regexp`*`")` | Full regular expression matching.<br><br>For more information on regular expressions, see Appendix A: *Regular Expressions* on page 119. | `if ( regmatch($enterprise, "^Acme Config:[0-9]") )` |
| `rtrim(`*`expression`*`)` | Removes white space from the right of an expression. | `$TrimNode = rtrim($Node)` |
| `scanformat(`*`expression`*`, "string")` | Converts the expression according to the following formats, similar to the `scanf` family of routines in C. Conversion specifications are:<br><br>`%%` - literal `%`; do not interpret<br><br>`%d` - matches an optionally signed decimal integer<br><br>`%u` - same as `%d`; no check is made for sign<br><br>`%o` - matches an optionally signed octal number<br><br>`%x` - matches an optionally signed hexadecimal number<br><br>`%i` - matches an optionally signed integer<br><br>`%e`, `%f`, `%g` - matches an optionally signed floating point number<br><br>`%s` - matches a string terminated by white space or end of string | `$element = "Foo is up in 15 seconds"`<br><br>`[$node, $state, $time] = scanformat($element, "%s is %s in %d seconds")`<br><br>This sets `$node`, `$state`, and `$time` to `Foo`, `up`, and `15`, respectively. |

Table 11: String Functions (3 of 3)

| Function | Description | Example |
|---|---|---|
| `num_returned_fields = split("string", destination_array, "field_separator")` | Separates the specified string into elements of the destination array.<br><br>The field separator separates the elements. The field separator itself is not returned. If you specify multiple characters in the field separator, when any combination of one or more of the characters is found in the string, a separation will occur.<br><br>Regular expressions are not allowed in the string or field separator. | `num_elements= split("bilbo:frodo:gandalf", names,":")`<br><br>creates an array with three entries:<br><br>`names[1] = bilbo`<br><br>`names[2] = frodo`<br><br>`names[3] = gandalf`<br><br>`num_elements` is set to `3`.<br><br>You must define the `names` array at the start of the rules file, before any processing statements, as described in *Using Arrays* on page 30. |
| `substr(expression,n, len)` | Extracts a substring, starting at the position specified in the second parameter, for the number of characters specified by the third parameter. | `$Substring = substr($Node,2,10)` |
| `upper(expression)` | Converts an expression to uppercase. | `$Node = upper($Node)` |

## Math Functions

You can use math functions to perform numeric operations on elements. Elements are stored as strings, so you must use these functions to convert them into integers before performing numeric operations. Table 12 describes the math functions supported in rules files.

Table 12: Math Functions

| Function | Description | Example |
|---|---|---|
| `int(numeric)` | Converts a numeric value into an integer. | `if int($PercentFull) > 80` |
| `real(numeric)` | Converts a numeric value into a real number. | `@DiskSpace= (real($diskspace)/real($total))*100` |

In the following example, the severity of an alert that monitors disk space usage is set depending on the amount of available disk space.

```
if (int($PercentFull) > 80 && int($PercentFull) <=85)
{
    @Severity=2
}
else if (int($PercentFull)) > 85 && int($PercentFull) <=90)
{
    @Severity=3
}
else if (int($PercentFull > 90 && int($PercentFull) <=95)
{
    @Severity=4
}
else if (int($PercentFull) > 95)
{
    @Severity=5
}
```

The percentage of disk space is not always provided in the event stream. The percentage of disk space can be calculated in the rules file as follows:

```
if (int($total) > 0)
{
    @DiskSpace=(100*int($diskspace))/int($total)
}
```

This can also be calculated using the `real` function:

```
if (int($total) > 0)
{
    @DiskSpace=(real($diskspace)/real($total))*100
}
```

The previous example can then be used to set the severity of the alert.

## Date and Time Functions

You can use date and time functions to obtain the current time or to perform date and time conversions. Times are specified in *Coordinated Universal Time* (*UTC*)—the number of elapsed seconds since 1 January 1970. Table 13 describes the date and time functions supported in rules files.

Table 13: Date and Time Functions (1 of 2)

| Function | Description | Example |
|---|---|---|
| `datetotime(`*`string`*`,` *`conversion_specification`*`)` | Converts a string into a time data type using the C library function `strptime()`. See the man page for `strptime` for more information. | `$Date = datetotime("Tue Dec 19 18:33:11 GMT 2000", "%a %b %e %T %Z %Y")` |

Table 13: Date and Time Functions (2 of 2)

| Function | Description | Example |
|---|---|---|
| `getdate()` | Takes no arguments and returns the current date as a date. | `$tempdate = getdate()` |
| `timetodate(`*`UTC, conversion_specification`*`)` | Converts a time value into a string using the C library function `strftime()`. See the man page for `strftime` for more information. | `@StateChange = timetodate ($StateChange, "%T, %D")` |

## Host and Process Utility Functions

You can use utility functions to obtain information about the environment in which the probe is running. Table 14 describes the host and process functions supported in rules files.

Table 14: Host and Process Utility Functions

| Function | Description | Example |
|---|---|---|
| `getenv(`*`string`*`)` | Returns the value of a specified environment variable. | `$My_OMNIHOME = getenv("OMNIHOME")` |
| `getpid()` | Returns the process ID of the running probe. | `$My_PID = getpid()` |
| `hostname()` | Returns the name of the host on which the probe is running. | `$My_Hostname = hostname()` |

## Lookup Table Operations

Lookup tables provide a way to add extra information in an event. A lookup table consists of a list of keys and values. It is defined with the `table` function and accessed using the `lookup` function.

The `lookup` function evaluates the expression in the keys of the named table and returns the associated value. If the key is not found, an empty string is returned. The `lookup` function has the following syntax:

```
lookup(expression,tablename)
```

You can create a lookup table directly in the rules file or in a separate file.

### Defining Lookup Tables in the Rules File

You can create a lookup table directly in the rules file using the following format:

```
table tablename={{"key","value"},{"key","value"}...}
```

**Note:** Table definitions must appear at the start of a rules file, before any processing statements. You can define multiple tables in a rules file. For changes to the lookup table to take effect, the probe must be forced to re-read the rules file. Refer to *Re-reading the Rules File* on page 16 for more information.

For example, to create a table that matches a node name to the department the node is in:

```
table dept={{"node1","Technical"},{"node2","Finance"}}
```

You can access this lookup table in the rules file as follows:

```
@ExtraChar=lookup(@Node,dept)
```

This example uses the `@Node` field as the key. If the value of the `@Node` field matches a key in the table, `@ExtraChar` is set to the corresponding value.

A lookup table can also have multiple columns. For example:

```
table example_table =
{{"key1", "value1", "value2", "value3"},
{"key2", "val1", "val2", "val3"}}
```

You can obtain values from a multiple value lookup table as follows:

```
[@Summary, @AlertKey, $error_code] = lookup("key1", "example_table")
```

## Defining Lookup Tables in a Separate File

Rather than including the lookup table directly in the rules file, you can create the table in a separate file. If you are specifying a single value, the file must be in the format:

```
key[TAB]value
key[TAB]value
```

To create a table in which the node name is matched to the department the node is in, use the following format:

```
node1[TAB]"Technical"
node2[TAB]"Finance"
```

The `table` function must appear at the start of a rules file, before any processing statements. Specify the full path to the lookup table file as follows:

```
table dept="/opt/netcool/omnibus/probes/solaris2/Dept"
```

You can then use this lookup table in the rules file as follows:

```
@ExtraChar=lookup(@Node,dept)
```

For multiple values, the format is:

```
key1[TAB]value1[TAB]value2[TAB]value3
key2[TAB]val1[TAB]val2[TAB]val3
```

You can also control how the probe processes external lookup tables with the `LookupTableMode` property, described in *Probe Properties and Command Line Options* on page 60. This property determines how errors are handled when external lookup tables do not have the same number of values on each line.

### Specifying Default Table Values

You can specify a default option to handle an event that does not match any of the key values in a table. The default statement must follow the specific table definition.

The following is an example for a table in the rules file:

```
table example_table =
{{"key1", "value1", "value2", "value3"},
{"key2", "val1", "val2", "val3"}}
default = {"defval1", "defval2", "defval3"}
```

The following is an example for a table in a separate file:

```
table dept="/opt/netcool/omnibus/probes/solaris2/Dept"
default = {"defval1", "defval2", "defval3"}
```

## Update on Deduplication Function

The deduplication process is managed by the ObjectServer, but it can be configured in the probe rules file. Use the `update` function to specify which fields of an alert are to be updated if the alert is deduplicated. This allows deduplication rules to be set on a per-alert basis.

The `update` function has the following syntax:

```
update(fieldname [, TRUE | FALSE ] )
```

If set to `TRUE`, update on deduplication is enabled. If set to `FALSE`, update on deduplication is disabled. The default is `TRUE`.

For example, to ensure that the `Severity` field is updated on deduplication, you can add the following to the rules file:

```
update(@Severity)
```

You can also disable update on deduplication for a specific field. For example:

```
update(@Severity, FALSE)
```

## Details Function

Details are extra elements created by a probe to display alert information that is not stored in a field of the `alerts.status` table. Alerts do not have detail information unless it is added.

Detail elements are stored in the ObjectServer details table, `alerts.details`. You can view details by double-clicking an alert in the event list.

You can add information to the details table using the `details` function. The detail information is added when an alert is inserted, but not if it is deduplicated.

The following example adds the elements `$a` and `$b` to the `alerts.details` table:

```
details($a,$b)
```

The following example adds all of the alert information to the `alerts.details` table:

```
details($*)
```

**Warning:** You should only use `$*` when you are debugging or writing rules files. After using `$*` for long periods of time, the ObjectServer tables will become very large and the performance of the ObjectServer will suffer.

In the following example, the `$Summary` element is compared to the strings `Incoming` and `Backup`. If there is no match, the `@Summary` field is set to the string `Please see details`, and all of the information for the alert is added to the details table:

```
if (match($Summary, "Incoming"))
{
     @Summary = "Received a call"
}
else if(match($Summary, "Backup"))
{
     @Summary = "Attempting to back up"
}
else
{
     @Summary = "Please see details"
```

```
    details($*)
}
```

## Message Logging Functions

You can use the `log` function to log messages during rules processing. You can also set a log level using the `setlog` function, and only messages equal to or above that level are logged.

There are five log levels: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `FATAL`, in order of increasing severity. For example, if the log level is set to `WARNING`, only `WARNING`, `ERROR`, and `FATAL` messages are logged, but if the logging is set to `ERROR` then only `ERROR` and `FATAL` messages are logged.

### Log Function

The `log` function sends a message to the log file.

The syntax is:

```
log([ DEBUG | INFO | WARNING | ERROR | FATAL ],"string")
```

**Note:** When a `FATAL` message is logged, the probe terminates.

### Setlog Function

The `setlog` function sets the minimum level at which messages are logged during rules processing. By default, the level for logging is `WARNING` and above.

The syntax is:

```
setlog([ DEBUG | INFO | WARNING | ERROR | FATAL ])
```

### Message Logging Example

The following is a sequence of logging functions executed in the rules file:

```
setlog(WARNING)
log(DEBUG,"A debug message")
log(WARNING,"A warning message")
setlog(ERROR)
log(WARNING,"Another warning message")
log(ERROR,"An error message")
```

This produces log output of:

```
A warning message
An error message
```

The `DEBUG` level message is not logged, because the logging setting is set higher than `DEBUG`. The second `WARNING` level message is not logged, because the preceding `setlog` function has set the log level higher than `WARNING`.

## Sending Alerts to Alternate ObjectServers and Tables

The `registertarget` function enables you to register one or more ObjectServers, and the corresponding tables, to which you may want to send alerts.You must register all potential targets at the start of a rules file, before any processing statements.

```
target_server = registertarget(server_name, backupserver_name, alerts_table [, details_
table ] )
```

In the following example, multiple targets are registered:

```
DefaultAlerts = registertarget( "TEST1", "", "alerts.status" )
HighAlerts = registertarget( "TEST2", "", "alerts.status" )
ClearAlerts = registertarget( "TEST3", "", "alerts.status" )
London = registertarget( "NCOMS", "", "alerts.london" )
```

**Note:** Regardless of the number of registered target ObjectServers, each alert can only be sent to one of them.

When you register more than one target, the one registered first is initially the default target. Unless another command overrides these settings, the alert destination following these register commands is the `alerts.status` table in the `TEST1` ObjectServer.

The `setdefaulttarget` function enables you to change the default ObjectServer to which alerts are sent when a target is not specified.

The `settarget` function enables you to specify the ObjectServer to which an alert will be sent without changing the default target.

You can change both the default ObjectServer and the ObjectServer to which specific alerts are sent, as shown in the following example:

```
# Once an event of Major severity or higher comes in,
# set the default ObjectServer to TEST2
if(int(@Severity) > 3)
{ setdefaulttarget(HighAlerts) }
# Send all clear events to TEST3
if (int(@Severity) = 0)
{ settarget(ClearAlerts) }
```

## Service Function

Use the `service` function to define the status of a service before alerts are forwarded to the ObjectServer. The status changes the color of the alert when it is displayed in the event list and Service windows.

The syntax is:

```
service(service_identifier, service_status)
```

The *`service_identifier`* identifies the monitored service, for example, `$host`.

Table 15 lists the service status levels.

Table 15: Service Function Status Levels

| Service Status Level | Definition |
|---|---|
| `BAD` | The service level agreement is not being met. |
| `MARGINAL` | There are some problems with the service. |
| `GOOD` | There are no problems with the service. |
| *No Level Defined* | The status of the service is unknown. |

### Service Function Example

If you want a Ping Probe to return a service status for each host it monitors, you can use the `service` function in the rules file to assign a service status to each alert. In the following example, a service status is assigned to each alert based on the value of the `status` element.

```
switch ($status)
{
case "unreachable":
@Severity = "5"
@Summary = @Node + " is not reachable"
@Type = 1
service($host, bad)              # Service Entry
case "alive":
@Severity = "3"
@Summary = @Node + " is now alive"
@Type = 2
service($host, good)             # Service Entry
case "noaddress":
@Severity = "2"
@Summary = @Node + " has no address"
service($host, marginal)         # Service Entry
case "removed":
@Severity = "5"
@Summary = @Node + " has been removed"
service($host, marginal)          # Service Entry
case "slow":
@Severity = "2"
@Summary = @Node + " has not responded within
trip time"
service($host, marginal)          # Service Entry
case "newhost":
@Severity = "1"
@Summary = @Node + " is a new host"
service($host, good)              # Service Entry
case "responded":
@Severity = "0"
@Summary = @Node + " has responded"
service($host, good)              # Service Entry
default:
@Summary = "Ping Probe error details: " + $*
@Severity = "3"
service($host, marginal)          # Service Entry
}
```

## Monitoring Probe Loads

To monitor load, it is necessary to obtain time measurements and calculate the number of events processed over time. The `updateload` function takes a time measurement each time it is called, and the `getload` function returns the load as events per second.

Each time the `updateload` function is executed, the current time stamp, recorded in seconds and microseconds, is added to the beginning of a series of time stamps. The remaining time stamps record the difference in time from the previous time stamp. For example, to take a time measurement and update a property called `load` with a new time stamp:

```
%load = updateload(%load)
```

**Tip:** Depending on the operating system, differing levels of granularity may be reported in time stamps.

You can specify a maximum time window for which samples are kept, and a maximum number of samples. By default, the time window is one second and the maximum number of samples is 50. You can specify the number of seconds for which load samples are kept and the maximum number of samples in the format:

```
time_window_in_seconds.max_number_of_samples
```

For example, to set or reset these values for the `load` property:

```
%load = "2.40"
```

When the number of seconds in the time window is exceeded, any samples outside of that time window are removed. When the number of samples reaches the limit, the oldest measurement is removed.

The `getload` function calculates the current load, returned as events per second. For example, to calculate the current load and assign it to a temporary element called `current_load`:

```
$current_load = getload(%load)
```

For an example of how to use the `load` function to monitor loads, see *Using Load Functions to Monitor Nodes* on page 58.

## 2.5 Testing Rules Files

You can test the syntax of a rules file using the Syntax Probe, `nco_p_syntax`. This is more efficient than actually running the probe to test that the syntax of the rules file is correct.

To run the Syntax Probe, enter:

```
nco_p_syntax -rulesfile /test/rules_file.rules
```

Use the `-rulesfile` command line option to specify the full path and file name of the rules file. The Syntax Probe always runs in debug mode. It connects to the ObjectServer, tests the rules file, displays any errors to the screen, and then exits. If no errors are displayed, the syntax of the rules file is correct.

## 2.6 Debugging Rules Files

When making changes to the rules file, adding new rules, or creating lookup tables it is useful to test the probe by running it in debug mode. This shows exactly how an event is being parsed by the probe and any possible problems with the rules file.

**Tip:** For changes to the rules file to take effect, the probe must be forced to re-read the rules file. Refer to *Re-reading the Rules File* on page 16 for more information.

To change the message level of a running probe to run in debug mode, issue the command `kill -USR2 pid` on the probe process ID (PID). Refer to the `ps` and `kill` man pages for more information.

Each time you issue the command `kill -USR2 pid`, the message level is cycled.

**Tip:** For CORBA probes, issue the command `kill` commands on the `nco_p_nonnative` process ID.

You also can set the probe to run in debug mode on the command line or in the properties file. To enable debug mode on the command line, enter the command:

```
$OMNIHOME/probes/arch/probename -messagelevel DEBUG -messagelog STDOUT
```

Alternatively, you can enter the following in the properties file:

```
MessageLevel: "DEBUG"
MessageLog: "STDOUT"
```

If you omit the `MessageLog` property or `-messagelog` command line option, the debug information is sent to the probe log file in the `$OMNIHOME/log` directory rather than to the screen.

## 2.7 Rules File Examples

The following sections show examples of typical rules file segments.

### Enhancing the Summary Field

This example rule tests if the `$trap-type` element is `Link-Up`. If it is, the `@Summary` field is populated with a string made up of `Link up on`, the name of the node from the record being generated, `Port`, and the value of the `$ifIndex` element:

```
if( match($trap-type,"Link-Up") )
{
    @Summary = "Link up on " + @Node + " Port " + $ifIndex
}
```

### Populating Multiple Fields

This example rule is similar to the previous rule except that the `@AlertKey` and `@Severity` fields are also populated:

```
if( match($trap-type, "Link-Up") )
{
    @Summary = "Link up on " + @Node + " Port " + $ifIndex
    @AlertKey = $ifIndex
    @Severity = 4
}
```

### Nested IF Statements

This example rule first tests if the trap has come from an `Acme` manager, and then tests if it is a `Link-Up`. If both conditions are met, the `@Summary` field is populated the values of the `@Node` field and `$ifIndex` and `$ifLocReason` elements:

```
if( match($enterprise,"Acme") )
{
    if( match($trap-type, "Link-Up") )
    {
    @Summary= "Acme Link Up on " + @Node + " Port " + $ifIndex +
    " Reason: "+$ifLocReason
    } }
```

## Regular Expression Match

This example rule tests for a line starting with `Acme Configuration:` followed by a single digit:

```
if (regmatch($enterprise,"^Acme Configuration:[0-9]"))
{
    @Summary="Generic configuration change for " + @Node
}
```

## Regular Expression Extract

This example rule tests for a line starting with `Acme Configuration`: followed by a single digit. If the condition is met, it extracts that single digit and places it in the `@Summary` field:

```
if (regmatch($enterprise,"^Acme Configuration:[0-9]"))
{
    @Summary="Acme error "+extract($enterprise,"^Acme Configuration:
    ([0-9])")+" on" + @Node
}
```

## Numeric Comparisons

This example rule tests the value of an element called `$freespace` as a numeric value by converting it to an integer and performing a numeric comparison:

```
if (int($freespace) < 1024)
{
    @Summary="Less than 1024K free on drive array"
}
```

## Simple Numeric Expressions

This example rule creates an element called `$tmpval`. The value of `$tmpval` is derived from the `$temperature` element, which is converted to an integer and then has 20 subtracted from it. The string element `$tmpval` contains the result of this calculation:

```
$tmpval=int($temperature)-20
```

## Strings and Numerics in One Expression

This example rule creates an element called `$Kilobytes`. The value of `$Kilobytes` is derived from the `$DiskSize` element, which is divided by 1024 before being converted to a string type with the letter `K` appended:

```
$Kilobytes = string(int($DiskSize)/1024) + "K"
```

## Using Load Functions to Monitor Nodes

This example shows how to measure load for each node that is generating events. If a node is producing more than five events per second, a warning is written to the probe log file. If more than 80 events per second are generated for all nodes being monitored by the probe, events are sent to an alternate ObjectServer and a warning is written to the probe log file.

```
# declare the ObjectServers HIGHLOAD and LOWLOAD
# declare the loads array
LOWLOAD = registertarget( "NCOMS_LOW", "", "alerts.status")
HIGHLOAD = registertarget( "NCOMS_HIGH", "", "alerts.status")
array loads;

# initialize array items with the number of seconds samples may span and
# number of samples to maintain.

if ( match("", loads[@Node]) ){
    loads[@Node] = "2.50"
}
if ( match("" , %general_load) ){
    %general_load="2.50"
}
loads[@Node] = updateload(loads[@Node])
%general_load=updateload(%general_load)
if ( int(getload(loads[@Node]) ) > 5 ){
    log(WARN, $Node + " is creating more than 5 events per second")
}
if ( int(getload(%general_load)) > 80){
   log(WARN, "Probe is creating more than 80 events per second - switching to HIGHLOAD")
    settarget(HIGHLOAD)
}
```

# Chapter 3: Probe Properties and Command Line Options

This chapter describes the properties and command line options common to all probes and TSMs. For the properties and command line options specific to a particular probe or TSM, refer to the individual guides for each probe and TSM available on the Micromuse Support Site.

For introductory information about probes, see Chapter 1: *Introduction to Probes* on page 9. For an introduction to probe properties, refer to *Properties File* on page 14.

This chapter contains the following section:

- *Probe Properties and Command Line Options* on page 60

## 3.1 Probe Properties and Command Line Options

The probe has default values for each property. In an unedited properties file, all properties are listed with their default values, commented out with a hash symbol (#) at the beginning of the line.

You can edit probe property values using a text editor. To override the default, change a setting in the properties file and remove the hash symbol. If you edit the properties file while the probe is running, the changes you make will take effect the next time you start the probe.

If you change a setting on the command line, this overrides both the default value and the setting in the properties file. To simplify the command you type to run the probe, add as many properties as possible to the properties file rather than using the command line options.

To run a probe, enter:

```
$OMNIHOME/probes/nco_p_probename [-option [ value ] ... ]
```

The `probename` is the abbreviated name of the probe you want to run. The `-option` is the command line option and `value` is the value you are setting the option to. Not every option requires you to specify a value.

If the `-name` command line option is specified, it determines the name used for the probe files described in Table 16:

Table 16: Names of Probe Files

| Type of File | Path and File Name |
|---|---|
| Properties File | `$OMNIHOME/probes/`*`arch`*`/`*`name`*`.props` |
| Rules File | `$OMNIHOME/probes/`*`arch`*`/`*`name`*`.rules` |
| Store and Forward File | `$OMNIHOME/var/`*`name`*`.store.`*`server`* |
| Message Log File | `$OMNIHOME/log/`*`name`*`.log` |

In these paths, *`arch`* is the name of the architecture on which the probe is installed; for example, `solaris2` when running on a Solaris system.

If the `-propsfile` command line option is specified, its value overrides the name setting for the properties file.

**Note:** Always read the guide that is specific to the probe you are running for additional configuration information. The individual probe guides are available on the Micromuse Support Site.

Table 17 lists the common properties and command line options available to all probes, and their default settings.

Table 17: Common Probe Properties and Command Line Options (1 of 7)

| Property | Command Line Option | Description |
|---|---|---|
| `AuthPassword` *`string`* | N/A | The password associated with the user name used to authenticate the probe when it connects to an ObjectServer running in secure mode. This password must be encrypted with the `nco_g_crypt` utility. The default is `''`.<br><br>Secure mode is described in *Secure Mode* on page 20. |
| `AuthUserName` *`string`* | N/A | A user name used to authenticate the probe when it connects to an ObjectServer running in secure mode. The default is `''`.<br><br>Secure mode is described in *Secure Mode* on page 20. |
| `AutoSAF 0 \| 1` | `-autosaf`<br><br>`-noautosaf` | Specifies whether or not automatic store and forward mode is enabled. By default, automatic store and forward mode is not enabled (`0`).<br><br>Store and forward mode is described in *Store and Forward Mode* on page 19.<br><br>**Note:** For failover to work, automatic store and forward must be enabled in addition to setting the `ServerBackup`, `NetworkTimeout`, and `PollServer` properties. |
| `BeatInterval` *`integer`* | `-beatinterval` *`integer`* | Specifies the heartbeat interval for peer-to-peer failover. The default is `2` seconds.<br><br>Peer-to-peer failover is described in *Peer-to-Peer Failover* on page 20. |
| `Buffering 0 \| 1` | `-buffer`<br><br>`-nobuffer` | Specifies whether or not buffering is used when sending alerts to the ObjectServer. By default, buffering is not enabled (`0`).<br><br>**Note:** All alerts sent to the same table are sent in the order in which they were processed by the probe. If alerts are sent to multiple tables, the order is preserved for each table, but not across tables. |
| `BufferSize` *`integer`* | `-buffersize` *`integer`* | Specifies the number of alerts the probe buffers. The default is `10` . |

Table 17: Common Probe Properties and Command Line Options (2 of 7)

| Property | Command Line Option | Description |
|---|---|---|
| N/A | `-help` | Displays the supported command line options and exits. |
| `LookupTableMode` *`integer`* | `-lookupmode` *`integer`* | Specifies how table lookups are performed. It can be set to `1`, `2`, or `3`. The default is `3`.<br><br>If set to `1`, all external lookup tables are assumed to have a single value column. Tabs are not used as column delimiters.<br><br>If set to `2`, all external lookup tables are assumed to have multiple columns. If the number of columns on each line is not the same, an error is generated that includes the file name and the line on which the error occurred.<br><br>If set to `3`, the rules engine attempts to determine the number of columns in the external lookup table. An error is generated for each line that has a different column count from the previous line. The error includes the file name and the line on which the error occurred.<br><br>Lookup tables are described in *Lookup Table Operations* on page 45. |
| `Manager` *`string`* | `-manager` *`string`* | Specifies the value of the `Manager` field for the alert. The default value is determined by the probe. |
| `MaxLogFileSize` *`integer`* | `-maxlogfilesize` *`integer`* | Specifies the maximum size the log file can grow to, in Bytes. The default is 1 MByte. Once the log file reaches the size specified, a second log file is started. When the second file reaches the maximum size, the first file is overwritten with a new log file and the process starts again. |
| `MaxRawFileSize` *`integer`* | N/A | Specifies the maximum size of the raw capture file, in KBytes. The default is unlimited (`-1`).<br><br>Raw capture mode is described in *Raw Capture Mode* on page 20. |
| `MaxSAFFileSize` *`integer`* | `-maxsaffilesize` *`integer`* | Specifies the maximum size the store and forward file can grow to, in Bytes. The default is 1 MByte.<br><br>Store and forward mode is described in *Store and Forward Mode* on page 19. |

Table 17: Common Probe Properties and Command Line Options (3 of 7)

| Property | Command Line Option | Description |
|---|---|---|
| `MessageLevel` *`string`* | `-messagelevel` *`string`* | Specifies the message logging level. Possible values are: `debug`, `info`, `warn`, `error`, and `fatal`. The default level is `warn`.<br><br>Messages that are logged at each level are listed below:<br><br>`fatal` - `fatal` only.<br><br>`error` - `fatal` and `error`.<br><br>`warn` - `fatal`, `error`, and `warn`.<br><br>`info` - `fatal`, `error`, `warn`, and `info`.<br><br>`debug` - `fatal`, `error`, `warn`, `info`, and `debug`. |
| `MessageLog` *`string`* | `-messagelog` *`string`* | Specifies where messages are logged. The default is `$OMNIHOME/log/`*`name`*`.log`.<br><br>`MessageLog` can also be set to `stdout` or `stderr`. |
| `Mode` *`string`* | `-mode` *`string`* | Specifies the role of the instance of the probe in a peer-to-peer failover relationship. The mode can be:<br><br>`master` - This instance is the master.<br><br>`slave` - This instance is the slave.<br><br>`standard` - There is no failover relationship.<br><br>The default is `standard`.<br><br>Peer-to-peer failover is described in *Peer-to-Peer Failover* on page 20. |
| `MsgDailyLog` `0 \| 1` | `-msgdailylog` `0 \| 1` | Specifies whether or not daily logging is enabled. By default, the daily backup of log files is not enabled (`0`).<br><br>**Note:** Because the time is checked regularly, when `MsgDailyLog` is set there is a slight reduction in performance. |
| `MsgTimeLog` *`string`* | `-msgtimelog` *`string`* | Specifies the time after which the daily log is created. The default is `0000` (midnight).<br><br>If `MsgDailyLog` set to `0`, this value is ignored . |

Table 17: Common Probe Properties and Command Line Options (4 of 7)

| Property | Command Line Option | Description |
|---|---|---|
| `Name` *`string`* | `-name` *`string`* | Specifies the name of the probe. This value determines the names of the properties file, rules file, message log file, and store and forward file, as listed in Table 16 on page 60. |
| `NetworkTimeout` *`integer`* | `-networktimeout` *`integer`* | Specifies a time in seconds after which the connection to the ObjectServer times out, should a network failure occur. The default is `0`, meaning that no timeout occurs.<br><br>If a timeout occurs, the probe attempts to connect to the secondary ObjectServer, identified by the `ServerBackup` property.<br><br>If a timeout occurs and no secondary ObjectServer is specified, the probe enters store and forward mode. Store and forward mode is described in *Store and Forward Mode* on page 19. |
| `PeerHost` *`string`* | `-peerhost` *`string`* | Specifies the hostname of the network element acting as the counterpart to this probe instance in a peer-to-peer failover relationship. The default is `localhost`.<br><br>Peer-to-peer failover is described in *Peer-to-Peer Failover* on page 20. |
| `PeerPort` *`integer`* | `-peerport` *`integer`* | Specifies the port through which the master and slave communicate in a peer-to-peer failover relationship. The default port is `99`.<br><br>Peer-to-peer failover is described in *Peer-to-Peer Failover* on page 20. |
| `PidFile` *`string`* | `-pidfile` *`string`* | Specifies the name of the file that stores the process ID for the device. The default is `$OMNIHOME/var/`*`name`*`.`*`pid`*, where *`name`* is the name of the probe and *`pid`* is the process ID. |

Table 17: Common Probe Properties and Command Line Options (5 of 7)

| Property | Command Line Option | Description |
|---|---|---|
| `PollServer` *`integer`* | `-pollserver` *`integer`* | If connected to a backup ObjectServer because failover occurred, a probe periodically attempts to reconnect to the primary ObjectServer. This property specifies the frequency in seconds at which the probe polls for the return of the primary ObjectServer. It does this by disconnecting and then reconnecting; to the primary ObjectServer if available, or to the secondary ObjectServer if the primary is not available. Polling is the only way the probe can determine if the primary ObjectServer is available. The default is `0`, meaning that no polling occurs.<br><br>Polling only occurs if the ObjectServer to which the probe is currently connected has the `BackupObjectServer` property, which designates a backup ObjectServer, set to `TRUE`.<br><br>**Note:** A probe may go into store and forward mode when the primary ObjectServer becomes unavailable. The first alert is not forwarded to the backup ObjectServer until the second alert opens the connection to the backup. If `PollServer` is set to less than the average time between alerts, the ObjectServer connection is polled before an alert is sent, and the probe does not go into store and forward mode. |
| `Props.CheckNames`<br>`TRUE \| FALSE` | N/A | When `TRUE`, the probe does not run if any specified property is invalid. The default is `TRUE`. |
| `PropsFile` *`string`* | `-propsfile` *`string`* | Specifies the name of the properties file. The default is `$OMNIHOME/probes/`*`arch`*`/`*`name`*`.props`, where *`name`* is the name of the probe and *`arch`* is the platform name of the architecture. |

Table 17: Common Probe Properties and Command Line Options (6 of 7)

| Property | Command Line Option | Description |
|---|---|---|
| `RawCapture 0 \| 1` | `-raw`<br><br>`-noraw` | Controls the raw capture mode. Raw capture mode is usually used at the request of Micromuse Support. By default, raw capture mode is disabled (`0`).<br><br>**Note:** Raw capture can generate a large amount of data. By default, the raw capture file can grow indefinitely, although you can limit the size using the `MaxRawFileSize` property. Raw capture can also slow probe performance due to the amount of disk activity required for a busy probe.<br><br>Raw capture mode is described in *Raw Capture Mode* on page 20. |
| `RawCaptureFile` *`string`* | `-capturefile` *`string`* | Specifies the name of the raw capture file. The default is `$OMNIHOME/var/`*`name`*`.cap`, where *`name`* is the name of the probe.<br><br>Raw capture mode is described in *Raw Capture Mode* on page 20. |
| `RawCaptureFileAppend 0 \| 1` | `-rawcapfileappend`<br><br>`-norawcapfileappend` | Specifies whether new data is appended to the existing raw capture file, instead of overwriting it. By default, the file is overwritten (`0`).<br><br>Raw capture mode is described in *Raw Capture Mode* on page 20. |
| `RetryConnectionCount` *`integer`* | N/A | Specifies the number of events the probe processes in store and forward mode before trying to reconnect to the ObjectServer. The default is `15`.<br><br>Store and forward mode is described in *Store and Forward Mode* on page 19. |
| `RetryConnectionTimeOut` *`integer`* | N/A | Specifies the number of seconds the probe processes events in store and forward mode before trying to reconnect to the ObjectServer. The default is `30`.<br><br>Store and forward mode is described in *Store and Forward Mode* on page 19. |

Table 17: Common Probe Properties and Command Line Options (7 of 7)

| Property | Command Line Option | Description |
|---|---|---|
| `RulesFile` *`string`* | `-rulesfile` *`string`* | Specifies the name of the rules file.<br><br>This can be a file name or URL that specifies a rules file located on a remote server that is accessible using the `http` protocol.<br><br>The default is `$OMNIHOME/probes/`*`arch`*`/`*`name`*`.rules`, where *`name`* is the name of the probe. |
| `SAFFileName` *`string`* | `-saffilename` *`string`* | Specifies the name of the store and forward file.<br><br>The default is `$OMNIHOME/var/`*`name`*`.store.`*`server`*, where *`name`* is the name of the probe and *`server`* is the name of the target ObjectServer.<br><br>If a name other than the default is specified, the `.`*`server`* extension is appended to the path and file name.<br><br>Store and forward mode is described in *Store and Forward Mode* on page 19. |
| `Server` *`string`* | `-server` *`string`* | Specifies the name of the ObjectServer or proxy server that alerts are sent to. The default is `NCOMS`.<br><br>For more information about configuring the ObjectServer or proxy server, see the Netcool/OMNIbus Administration Guide. |
| `ServerBackup` *`string`* | N/A | Specifies a secondary ObjectServer should the primary ObjectServer connection fail. If `NetworkTimeout` is set, use `ServerBackup` to identify a secondary ObjectServer. |
| `StoreAndForward`<br>`0 \| 1` | `-saf`<br><br>`-nosaf` | Controls the store and forward operations. By default, store and forward mode is enabled (`1`).<br><br>Store and forward mode is described in *Store and Forward Mode* on page 19. |
| N/A | `-version` | Displays version information and exits. |

# Chapter 4: Introduction to Gateways

This chapter introduces gateways, their key features, and how to use them. It also describes the types of gateways, their components, and how to run them.

For information about commands common to all gateways and `nco_gate` command line options, refer to Chapter 5: *Gateway Commands and Command Line Options* on page 95.

For descriptions of gateway error messages, refer to Appendix D: *Gateway Error Messages* on page 155.

For information about specific gateways, refer to the documentation available for each gateway on the Micromuse Support Site. Some gateways have a different architecture than that described in this chapter, and do not use `nco_gate`.

This chapter contains the following sections:

## 4.1 Introduction to Gateways

Netcool/OMNIbus gateways enable you to exchange alerts between ObjectServers and complementary third-party applications, such as databases and helpdesk or Customer Relationship Management (CRM) systems.

You can use gateways to replicate alerts or to maintain a backup ObjectServer. Application gateways enable you to integrate different business functions. For example, you can configure a gateway to send alert information to a helpdesk system. You can also use a gateway to archive alerts to a database.

Figure 3 shows an example gateway architecture.

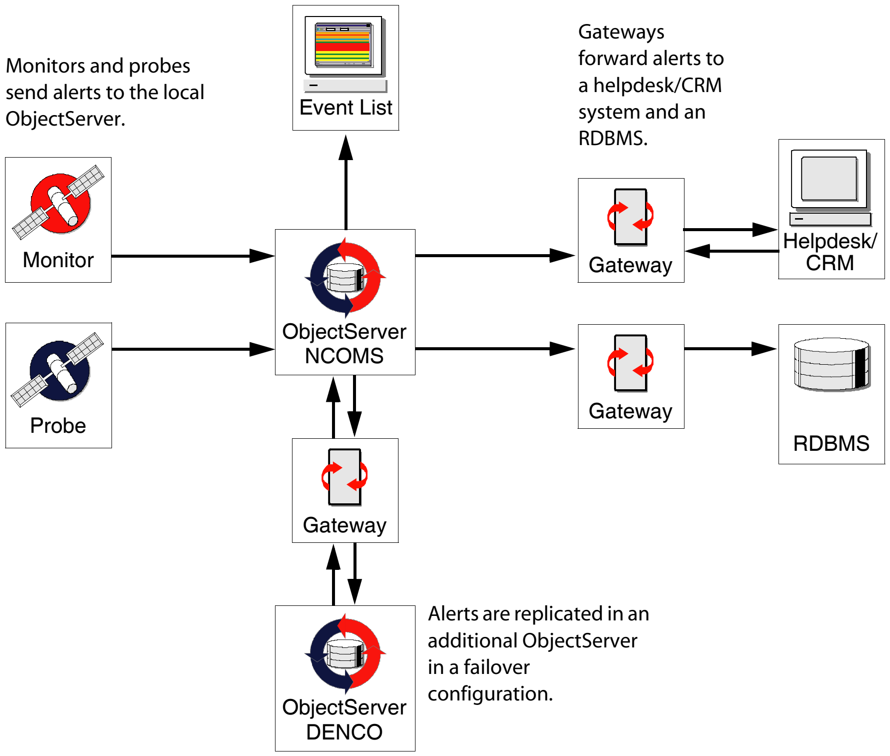

Monitors and probes send alerts to the local ObjectServer.
Event List
Gateways forward alerts to a helpdesk/CRM system and an RDBMS.
Monitor
Gateway
Helpdesk/ CRM
ObjectServer NCOMS
Probe
Gateway
RDBMS
Gateway
Alerts are replicated in an additional ObjectServer in a failover configuration.
ObjectServer DENCO

Figure 3: Gateways in the Netcool/OMNIbus Architecture

In the example in Figure 3, gateways are used for a variety of purposes:

- The ObjectServer Gateway enables you to replicate alerts between ObjectServers in a failover configuration.
- RDBMS gateways enable you to store critical alerts in a database so you can analyze network performance.
- Helpdesk gateways enable you to integrate the NOC and the helpdesk by converting trouble tickets to alerts and alerts to trouble tickets.

Once a gateway is correctly installed and configured, the transfer of alerts is transparent to operators. For example, alerts are forwarded from an ObjectServer to a database automatically without user intervention.

## 4.2 Types of Gateways

There are two main types of gateways:

- Unidirectional (archive) gateways
- Bidirectional (synchronization) gateways

*Unidirectional* gateways only allow alerts to flow in one direction. Changes made in the source ObjectServer are replicated in the destination ObjectServer or application, but changes made in the destination ObjectServer or application are not replicated in the source ObjectServer. They can be considered as *archiving* tools.

*Bidirectional* gateways allow alerts to flow from the source ObjectServer to the target ObjectServer or application and also allow feedback to the source ObjectServer. In a bidirectional gateway configuration, changes made to the contents of a source ObjectServer are replicated in a destination ObjectServer or application, and the destination ObjectServer or application replicates its alerts in the source ObjectServer. They can be considered as *synchronization* tools.

Gateways are able to send alerts to a variety of targets:

- Another ObjectServer
- A database
- A helpdesk application
- Other applications or devices

ObjectServer gateways are used to exchange alerts between ObjectServers. This is useful when you want to create a distributed installation, or when you want to install a backup ObjectServer.

Database gateways are used to store alerts from an ObjectServer. This is useful when you want to keep a historical record of the alerts forwarded to the ObjectServer.

Helpdesk gateways are used to integrate Netcool/OMNIbus with a range of helpdesk systems. This is useful when you want to correlate the trouble tickets raised by your customers with the networks and systems you are using to provide their services.

Other gateways are specialized applications that forward ObjectServer alerts to other applications or devices (for example, a flat file or socket).

**Note:** Only gateways that send alerts to certain targets can be bidirectional. For more information, refer to the individual gateway guides available on the Micromuse Support Site.

## 4.3 ObjectServer Gateways

This section provides a brief overview of the unidirectional and bidirectional ObjectServer gateways.

**Note:** ObjectServer gateways have been revised for Netcool/OMNIbus v7 and do not use the `nco_gate` binary which is common to many other gateways.

The unidirectional and bidirectional ObjectServer gateways are described in detail in the guide for the ObjectServer Gateway, available on the Micromuse Support Site.

### Unidirectional ObjectServer Gateway

A unidirectional ObjectServer Gateway allows alerts to flow from a source ObjectServer to a destination ObjectServer. Changes made in the source ObjectServer are replicated in the destination ObjectServer, but changes made in the destination ObjectServer are not replicated in the source ObjectServer.

Figure 4 shows the configuration of a unidirectional ObjectServer Gateway:

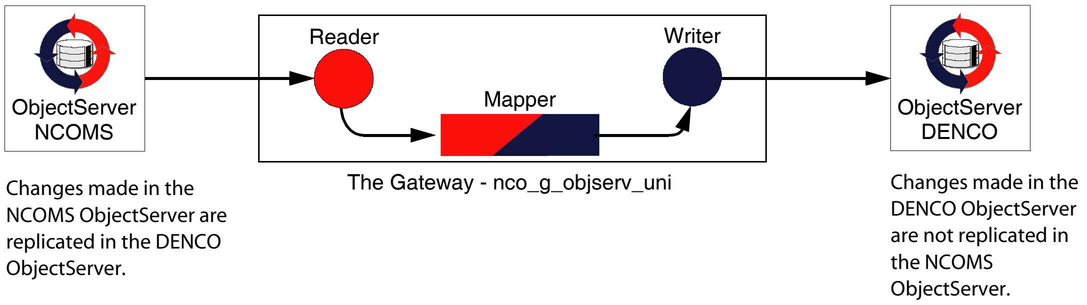

ObjectServer
NCOMS
Reader
Writer
Mapper
ObjectServer
DENCO
The Gateway - nco_g_objserv_uni
Changes made in the NCOMS ObjectServer are replicated in the DENCO ObjectServer.
Changes made in the DENCO ObjectServer are not replicated in the NCOMS ObjectServer.

Figure 4: Unidirectional ObjectServer Gateway

### Bidirectional ObjectServer Gateway

A bidirectional ObjectServer Gateway allows alerts to flow from a source ObjectServer to a destination ObjectServer. Changes made to the contents of a source ObjectServer are replicated in a destination ObjectServer, and the destination ObjectServer replicates its alerts in the source ObjectServer. This enables you, for example, to maintain a system with two ObjectServers configured as a failover pair.

Figure 5 shows the configuration of a bidirectional ObjectServer Gateway:

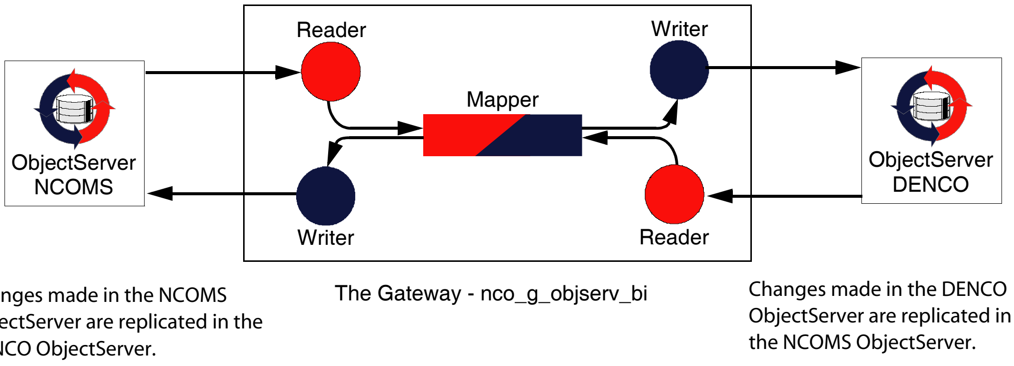

Reader
Writer
Mapper
ObjectServer NCOMS
ObjectServer DENCO
Writer
Reader
Changes made in the NCOMS ObjectServer are replicated in the DENCO ObjectServer.
The Gateway - nco_g_objserv_bi
Changes made in the DENCO ObjectServer are replicated in the NCOMS ObjectServer.

Figure 5: Bidirectional ObjectServer Gateway

## ObjectServer Gateway Writers and Failback (Alert Replication Between Sites)

Failover occurs when a gateway loses its connection to the primary ObjectServer; this allows the gateway to connect to a backup ObjectServer. Failback functionality allows the gateway to reconnect to the primary ObjectServer when it becomes active again.

Because bidirectional ObjectServer gateways are used to resynchronize failover pairs, failback is automatically disabled. This is because one half of the gateway can legitimately be connected to a backup server and so should not be forced to keep failing back to the primary ObjectServer.

However, if a bidirectional gateway is being used to share data between two separate sites, and each site has a failover pair operating, you can manually enable failback on each server. When enabled, the writer automatically enables failback on its counterpart reader.

For more information about ObjectServer gateways and how to enable failback, see the guide for the ObjectServer Gateway, available on the Micromuse Support Site.

## 4.4 Database, Helpdesk, and Other Gateways

Most database, helpdesk, and other gateways are based on the generic gateway binary `nco_gate`. Additional modules handle the communication with the target applications, devices, or files.

**Note:** While all gateways have the components described in this section, some gateways have a different architecture than that described, and do not use `nco_gate`. For more information, refer to the individual gateway guides available on the Micromuse Support Site.

### Gateway Components

Gateways have *reader* and *writer* components. Readers extract alerts from the ObjectServer. Writers forward alerts to another ObjectServer or to other applications. There is only one type of reader, but there are various types of writers depending on the destination application.

*Routes* specify the destination to which a reader forwards alerts. One reader can have multiple routes to different writers, and one writer can have multiple routes from different readers.

Gateway *filters* and *mappings* configure alert flow. Filters define the types of alerts that can be passed through a gateway. Mappings define the format of these alerts.

Readers, writers, routes, filters, and mappings are defined in the gateway configuration file, described in *Gateway Configuration* on page 79.

### Unidirectional Gateways

A simple example of a gateway is the Flat File Gateway. This is a unidirectional gateway that reads alerts from an ObjectServer and writes them to a flat file. This example architecture is shown in Figure 6.

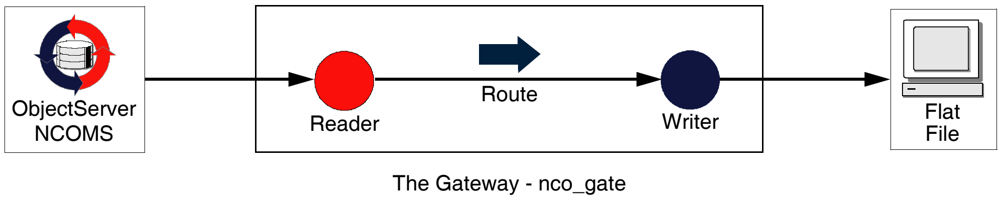


Figure 6: Example Flat File Gateway Architecture

## Bidirectional Gateways

In a bidirectional gateway configuration, changes made to the alerts in a source ObjectServer are replicated in a destination application, and the destination application replicates changes to its alerts in the source ObjectServer. This enables you, for example, to raise trouble tickets in a helpdesk system for certain alerts. Changes made to the tickets in the helpdesk system can then be sent back to the ObjectServer.

Bidirectional gateways have a similar configuration to unidirectional gateways, with an additional `COUNTERPART` attribute for the writers. The `COUNTERPART` attribute defines a link between a gateway's writer and reader.

Figure 7 shows an example bidirectional gateway configuration.

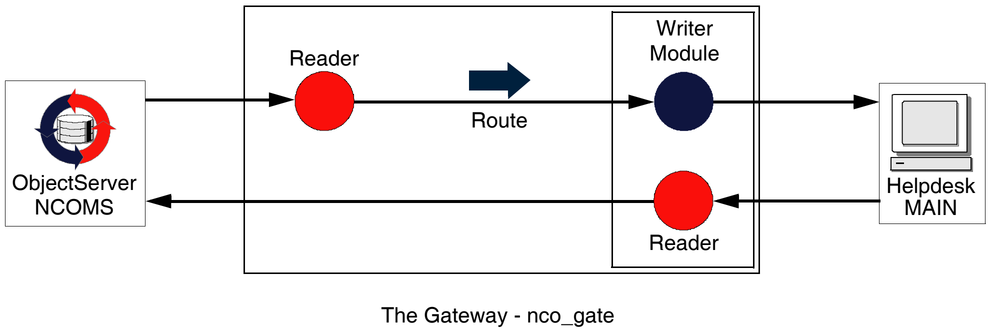


Figure 7: Bidirectional Helpdesk Gateway

### The Reader

A reader extracts alerts from an ObjectServer. There is only one type of reader: the ObjectServer reader. When the reader starts, the gateway attempts to open a connection to the source ObjectServer. If the gateway succeeds in opening the connection, it immediately starts to read alerts from the ObjectServer.

### Writer Modules

Writer modules manage communications between gateways and third-party applications, and format the alert correctly for entry into the application.

The writer module generates log files which can help debug the gateway. The log files are described in *Gateway Debugging* on page 91.

Communication between the writer module and the third-party application uses helper applications, which interact directly with the application through its APIs or other interfaces. These processes are transparent to the user (though they are visible using the `ps` command or similar utility).

The writer module uses a reference number cache to track the alerts and their associated reference number in the target application. For each alert, the cache stores the following:

- The serial number of the alert
- A reference number from the target application (for example, Clarify Cases or ServiceCenter Tickets)

When a ticket is raised in response to an alert, the writer module enters the reference number in the cache and returns it to the ObjectServer where the alert is updated to include the reference number.

Figure 8 shows a simplified example of the writer module architecture.

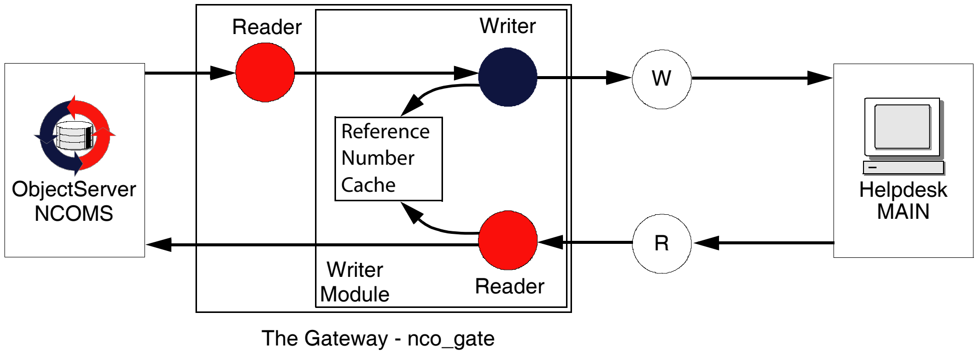


Figure 8: Reader/Writer Module Architecture

## Routes

Routes create the link between the source reader and the destination writer. Any alerts received by the source ObjectServer are read by the reader, passed through the route to the writer, and written into the destination ObjectServer or application.

## Alert Updates from the Helpdesk

When the helpdesk operator makes additional changes to the ticket, these are forwarded to the gateway which executes the corresponding action `.sql` file to update the alert in the ObjectServer. Typically the following action `.sql` files are provided:

- `open.sql`
- `update.sql`
- `journal.sql`
- `close.sql`

For detailed information on configuring alert updates from the helpdesk, see the gateway guide for the type of gateway that you are using. These are available on the Micromuse Support Site.

## 4.5 Gateway Configuration

The configuration file defines the operation of the gateway using:

- Readers
- Writers
- Routes
- Mappings
- Filters

This section describes the gateway configuration file and the commands used in it to define the operation of the gateway.

**Note:** ObjectServer gateways have been revised for Netcool/OMNIbus v7. The configuration file which is common to other gateways is replaced with a properties file. For more information, see the guide for the ObjectServer gateway.

### Gateway Configuration File

When a gateway is started, it processes the commands in the configuration file. This defines the connections between the source ObjectServer and the alert destinations.

Every gateway has a configuration file, with the extension `.conf`. The default gateway configuration file is:

```
$OMNIHOME/etc/NCO_GATE.conf
```

Use the `-config` command line option to specify the full path and name of an alternate configuration file. For example, to run a helpdesk gateway with a configuration file named `HDESK.conf`, enter:

```
$OMNIHOME/bin/nco_gate -config $OMNIHOME/etc/HDESK.conf
```

### Reader Commands

A reader extracts alerts from an ObjectServer. There is only type of reader: the ObjectServer reader. Readers are started using the `START READER` command, which defines the name of the reader and the name of the ObjectServer from which to read.

For example, to start a reader for the NCOMS ObjectServer shown in Figure 6 on page 75, add the following command to the configuration file:

```
START READER NCOMS_READ CONNECT TO NCOMS;
```

Once this command has been issued, the reader is started and the gateway attempts to open a connection to the source ObjectServer. If the gateway succeeds in opening the connection, it immediately starts to read alerts from the ObjectServer. For the reader to forward these alerts to their destination, you must define an associated route and writer.

The START READER command is described in more detail in *Reader Commands* on page 98.

## Writer Commands

Writers send the alerts acquired by a reader to the destination application or ObjectServer. Writers are created using the START WRITER command, which defines the name of the writer and the information that allows it to connect to its destination.

For example, to create the writer for the Flat File Gateway shown in Figure 6 on page 75, add the following command to the configuration file:

```
START WRITER FILE_WRITER

(
        TYPE = FILE,
        REVISION = 1,
        FILE = '/tmp/omnibus/log/NCOMS_alert.log',
        MAP = FILE_MAP,
        INSERT_HEADER = 'INSERT: ',
        UPDATE_HEADER = 'UPDATE: ',
        DELETE_HEADER = 'DELETE: ',
        START_STRING = '"',
        END_STRING = '"',
        INSERT_TRAILER = '\n',
        UPDATE_TRAILER = '\n',
        DELETE_TRAILER = '\n'
);
```

Once the START WRITER command has been issued, the gateway attempts to establish the connection to the alert destination (either an application or another ObjectServer). The writer sends alerts received from the source ObjectServer until the STOP WRITER command is issued.

The START WRITER command is described in more detail in *Writer Commands* on page 101.

## Route Commands

Routes create the link between readers and writers. Routes are created using the ADD ROUTE command. This command defines the name of the route, the source reader, and the destination writer.

For example, to create the route between the NCOMS ObjectServer reader and the writer for the Flat File Gateway shown in Figure 6 on page 75, add the following command to the configuration file:

```
ADD ROUTE FROM NCOMS_READ TO FILE_WRITER;
```

Once this command is issued, the connection between a reader and writer is established. Any alerts received by the source ObjectServer are read by the reader, passed through the route to the writer, and written into the destination ObjectServer or application.

The ADD ROUTE command is described in more detail in *Route Commands* on page 110.

## Mapping Commands

Mappings define how alerts received from the source ObjectServer should be written to the destination ObjectServer or application. Each writer has a different mapping which is defined using the CREATE MAPPING command.

For example, to create the mapping between the ObjectServer reader and the writer for the Flat File Gateway shown in Figure 6 on page 75, add the following command to the configuration file:

```
CREATE MAPPING FILE_MAP
(
     ''= '@Identifier',
     ''= '@Serial',
     ''= '@Node' ,
     ''= '@Manager',
     ''= '@FirstOccurrence' CONVERT TO DATE,
     ''= '@LastOccurrence'  CONVERT TO DATE,
     ''= '@InternalLast'    CONVERT TO DATE,
     ''= '@Tally',
     ''= '@Class',
     ''= '@Grade',
     ''= '@Location',
     ''= '@ServerName',
     ''= '@ServerSerial'
);
```

In this example, the mapping name is FILE_MAP.

Each line between the parentheses defines how the gateway writes alerts into the file. For the Flat File Gateway, the CREATE MAPPING command defines the fields from which data is written into each alert in the output file. The alert fields from the source ObjectServer are represented by the @ symbol.

The following example shows INSERT and UPDATE commands using the FILE_MAP mapping shown above.

```
INSERT: "Downlink6LinkMon4Link",127,"sfo4397","Netcool Probe",12/05/03
15:39:23,12/05/03 15:39:23,12/05/03 15:30:53,1,3300,0,"","NCOMS",127
UPDATE: "muppetMachineMon2Systems",104,"sfo4397","Netcool Probe",12/05/03
12:29:34,12/05/03 15:40:06,12/05/03 15:31:36,11,3300,0,"","NCOMS",104
UPDATE: "muppetMachineMon4Systems",93,"sfo4397","Netcool Probe",12/05/03
12:29:11,12/05/03 15:40:35,12/05/03 15:32:05,12,3300,0,"","NCOMS",93
```

Other gateways (with the exception of the Socket Gateway) require a field in the target to be specified for each source ObjectServer field. For example, in the Gateway for Remedy ARS, source ObjectServer fields are mapped to Remedy ARS fields, which are identified with long integer values rather than field names. In the following example, the ARS field 536870913 maps to the Serial field from the ObjectServer:

```
536870913 = '@Serial' ON INSERT ONLY
```

The ON INSERT ONLY clause controls when the field is updated. Fields with the ON INSERT ONLY clause are only forwarded once, when the alert is created for the first time in the ObjectServer. Fields that do not have the ON INSERT ONLY clause are updated each time the alert changes.

The CREATE MAPPING command is described in more detail in *Mapping Commands* on page 105.

## Filter Commands

You may not always want to send all of the alerts read by a reader to the destination application. For example, you may only want to send alerts that have a severity level of Critical. Filters define which of the alerts read by the ObjectServer reader should be forwarded to the destination.

You create filters using the CREATE FILTER command and apply them using the START READER command. For example, to create a filter that only forwards critical alerts to the destination application or ObjectServer, add the following command to the configuration file:

```
CREATE FILTER CRITONLY AS 'Severity = 5';
```

This command creates a filter named CRITONLY, which only forwards alerts with a severity level of Critical (5).

**Note:** To perform string comparisons with filters, you must escape the quotes in the CREATE FILTER command with backslashes. For example, to create a filter that only forwards alerts from a node called fred, the CREATE FILTER command is:

```
CREATE FILTER FREDONLY AS 'NODE = \'fred\'';
```

To apply the filter to the ObjectServer reader shown in Figure 6 on page 75, add the following command to the configuration file:

```
START READER NCOMS_READ CONNECT TO NCOMS USING FILTER CRITONLY;
```

The `CREATE FILTER` command is described in more detail in *Filter Commands* on page 108.

## Creating Multiple Filters and Multiple Readers

If you need more than one filter for the same ObjectServer, you can create multiple readers for it. For example, to create a reader that forwards all critical alerts and another that forwards everything else, use the following commands:

```
CREATE FILTER CRITONLY AS 'Severity = 5';
CREATE FILTER NONCRIT AS 'Severity < 5';
START READER CRIT_NCOMS CONNECT TO NCOMS USING FILTER CRITONLY;
START READER NONCRIT_NCOMS CONNECT TO NCOMS USING FILTER NONCRIT;
```

## Loading Filters Created Using Filter Builder

You can load and use filters created in the Filter Builder. For example:

```
LOAD FILTER FROM '/usr/filters/myfilt.elf';
```

This command loads the file `/usr/filters/myfilt.elf` as a filter. This filter name is defined by the Filter Builder `Name` field.

**Note:** The `Name` field must be alphabetical and must not contain spaces.

For more information about the Filter Builder, refer to the Netcool/OMNIbus User Guide.

## 4.6 Running a Gateway

A gateway requires an entry in the Server Editor, as described in the Netcool/OMNIbus Installation and Deployment Guide.

You must also create your configuration file, described in *Gateway Configuration* on page 79.

Once you have defined the gateway communications and created your configuration file, you can run the gateway.

**Note:** Some gateways have a different architecture than that described in this chapter, and do not use `nco_gate`. For information about specific gateways, refer to the documentation available for each gateway on the Micromuse Support Site.

### Running a Gateway on UNIX

To run a gateway with a default configuration, enter:

```
$OMNIHOME/bin/nco_gate
```

This runs a gateway with the default name `NCO_GATE` and the default configuration file `$OMNIHOME/etc/NCO_GATE.conf`.

To run a gateway with your own configuration, use command line options, as described in *Gateway Command Line Options* on page 96. For example:

```
$OMNIHOME/bin/nco_gate -name ORA1 -config $OMNIHOME/etc/RDBMS.conf
```

This runs a gateway named `ORA1` using a configuration file named `RDBMS.conf`.

Gateways should be configured to run under process control. Process control is described in the Netcool/OMNIbus Administration Guide.

### Running a Gateway on Windows

Gateways on Windows can be run as console applications or as services.

#### Running a Gateway as a Console Application

To run a gateway as a console application, enter the following at the command line:

```
%OMNIHOME%\bin\nco_gate
```

This runs a gateway with the default name `NCO_GATE` and the default configuration file `%OMNIHOME%\etc\NCO_GATE.conf`.

To run a gateway as a console application with your own configuration, use the command line options, as described in *Gateway Command Line Options* on page 96. For example:

```
%OMNIHOME%\bin\nco_gate -name ORA1 -config %OMNIHOME%\etc\RDBMS.conf
```

This runs a gateway named `ORA1` using a configuration file named `RDBMS.conf`.

## Running a Gateway as a Service

To run a gateway as a service, use the `-install` command line option.

You can configure how gateways are started using the *Services* window as follows:

1. Click **Start**→**Settings**→**Control Panel**. The Control Panel is displayed.
2. Double-click the **Admin Tools** icon, then double-click the **Services** icon. The *Services* window is displayed.

   The *Services* window lists all of the Windows services currently installed on your machine. All Netcool/OMNIbus services start with `NCO`.
3. Use the *Services* window to start and stop Windows services. Define whether the service is started automatically when the machine is booted by clicking the **Startup** button.

## 4.7 Configuring Gateways Interactively

You can change the configuration of a gateway while it is running using the SQL interactive interface. The SQL interactive interface is described in the Netcool/OMNIbus Administration Guide.

**Note:** If you are running a gateway on UNIX, you must be a member of the UNIX user group that is allowed to log into a gateway. By default, this is the `ncoadmin` user group. You may need to ask your system administrator to create this group. The `-admingroup` command line option is described in *Gateway Command Line Options* on page 96.

Use the SQL interactive interface to connect to a gateway as a specific user. For example:

Table 18: Connecting to the Gateway Using the SQL Interactive Interface

| On... | Enter the following command... |
|---|---|
| UNIX | `$OMNIHOME/bin/nco_sql -server` *`servername`* `-user` *`username`* |
| Windows | `%OMNIHOME%\bin\redist\isql.exe -s` *`servername`* `-u` *`username`* |

In these commands, *`servername`* is the name of the gateway and *`username`* is a valid user name. If you do not specify a user name, the default is the user running the command.

You are prompted to enter a password. On UNIX, the default is to enter your UNIX password. To authenticate users using other methods, use the `-authenticate` command line option, described in *Gateway Command Line Options* on page 96.

After connecting with a user name and password, a numbered prompt is displayed.

```
1>
```

You can enter commands to configure the gateway dynamically. The following example shows a session in which new routes are added:

```
$ nco_sql -server NCO_GATE
Password:
User 'admin' logged in.
1> ADD ROUTE FROM DENCO_READ TO ARS_WRITER;
2> ADD ROUTE FROM DENCO_READ TO OS_WRITER;
3> go

1>
```

If you want to disable interactive configuration, add the following line to the end of the gateway configuration file:

```
SET CONNECTIONS FALSE;
```

## Saving Configurations Interactively

You can save the interactive gateway configuration with the command:

```
SAVE CONFIG TO 'filename';
```

In this command, *`filename`* is the name of a file on a local file system.

You can then use the saved configuration file for other gateways.

## Dumping and Loading Gateway Configurations Interactively

You can load gateway configurations interactively.

First stop any running readers and writers manually with the `STOP` command. Then use the `DUMP CONFIG` command to discard the current configuration.

The `DUMP CONFIG` command will not discard the configuration if any readers and writers are running or if the configuration has been changed interactively, unless you use the `FORCE` option. To determine if the configuration has been changed interactively, use the `SHOW SYSTEM` command, described in *SHOW SYSTEM* on page 115.

Refer to *Configuration Commands* on page 112 for more information.

Once you have dumped the configuration, you can load a new configuration with the command:

```
LOAD CONFIG FROM 'filename';
```

In this command, *`filename`* is the name of a file on a local file system.

## 4.8 Gateway Features

This section describes some of the key features of gateway operation.

### Store and Forward Mode

If there is a problem with the gateway target, the ObjectServer and database writers can continue to run using store and forward mode.

When the writer detects that the target ObjectServer or database is not present or is not functioning (usually because the writer is unable to write an alert), it switches into store mode. In this mode, the writer stores everything it would normally send to the database in a file named:

`$OMNIHOME/var/`*`writername.destserver`*`.store`

In this file name, *`writername`* is the name of the writer and *`destserver`* is the name of the server to which the gateway is attempting to send alerts.

When the gateway detects that the destination server is back on line, it switches into forward mode and sends the alert information held in the `.store` file to the destination server. Once all of the alerts in the `.store` file have been forwarded, the writer returns to normal operation.

Store and forward mode only works when a connection to the ObjectServer or database destination has been established, used, and then lost. If the destination server is not running when the gateway starts, store and forward mode is not triggered and the gateway terminates.

If the gateway connects to the destination ObjectServer and a store and forward file already exists, the gateway replays the contents of the store and forward file before it sends new alerts.

Store and forward mode is configured using the attributes `STORE_AND_FORWARD` and `STORE_FILE`.

**Note:** Refer to the individual gateway documentation to determine whether an individual gateway supports store and forward mode. Store and forward does not work with bidirectional gateway configurations, with the exception of the bidirectional ObjectServer Gateway.

### Secure Mode

You can run the ObjectServer in secure mode. When you start the ObjectServer using the `-secure` command line option, the ObjectServer authenticates probe, gateway, and proxy server connections by requiring a user name and encrypted password. When a connection request is sent, the ObjectServer issues an authentication message. The probe, gateway, or proxy server must respond with the correct user name and password. If the user name and password combination is incorrect, the ObjectServer issues an error message and rejects the connection.

If the ObjectServer is not running in secure mode, probe, gateway, and proxy server connection requests are not authenticated.

When connecting to a secure ObjectServer, the gateway must have the `AUTH_USER` and `AUTH_PASSWORD` commands in the gateway configuration file. You can choose any valid user name for the `AUTH_USER` gateway command. To generate the encrypted `AUTH_PASSWORD`, use the `nco_g_crypt` utility, described in the Netcool/OMNIbus Administration Guide. The command takes the unencrypted password and displays the encrypted password to be entered for the `AUTH_PASSWORD` command.

**Note:** If you are connecting to a version 3.6 ObjectServer, use the `nco_crypt` utility, rather than the `nco_g_crypt` utility, to encrypt the password. If you are using PAM for authorization in your version 3.6 ObjectServer, you must use a plain text password.

The `AUTH_USER` and `AUTH_PASSWORD` commands must precede any reader commands in the gateway configuration file. Before running the gateway, add the user name and corresponding encrypted password to the configuration file, for example:

```
AUTH_USER 'Gate_User'
AUTH_PASSWORD 'Crypt_Password'
```

**Note:** ObjectServer gateways have been revised for Netcool/OMNIbus v7. To connect to a secure ObjectServer from the ObjectServer Gateway, the gateway properties for the user name and password must be set. For more information, see the guide for the ObjectServer Gateway.

## Encrypting Target System Passwords

You can also use the `nco_g_crypt` utility to encrypt plain text login passwords. The gateways use these encrypted passwords to log into their target systems. Encrypted passwords are decoded by the gateway before they are used to log in to the target system.

The user name and encrypted password are stored in the `USERNAME` and `PASSWORD` attributes in the gateway writer.

**Note:** If you are using a helpdesk gateway, substitute `USER` for `USERNAME`.

To encrypt a plain text password for a gateway's target system:

1. Use the `nco_g_crypt` utility to obtain an encrypted version of the password.
2. Update the gateway writer in the gateway configuration file by copying the user name into the `USERNAME` attribute value and the encrypted password created in step 1 into the `PASSWORD` attribute value.

For example:

```
START WRITER SYBASE_WRITER
(
    TYPE = SYBASE,
    REVISION = 1,
    SERVER = DARKSTAR,
    MAP = SYBASE_MAP,
    USER = 'SYSTEM',
    PASSWORD = 'MKFGHIFE',
    FORWARD_DELETES = TRUE
);
```

3. Run the gateway.

## 4.9 Gateway Debugging

When debugging you should initially check the log file:

```
$OMNIHOME/log/NCO_GATENAME.log
```

Where *GATENAME* is the name of the gateway.

You might receive an error message such as the following:

```
error in srv_select () - file descriptor x is no longer active!
```

This type of error message indicates that the gateway has aborted because one of the reader or writer modules failed. In this case, check the following log files:

```
NCO_GATE_XRWY_WRITE.log
```

or

```
NCO_GATE_XRWY_READ.log
```

Where *X* identifies the name of the gateway and *Y* identifies the version of that gateway.

## 4.10 Other Gateway Writers and Failback

The ObjectServer reader can fail over and fail back between source ObjectServers without shutting down. This ability is not supported by all gateway writers. If a writer does not support this mode of failback and failover, the writer, on detection of the reader failover/failback, will shut down the gateway and rely on the process agent to restart the gateway.

Writers that support reader failover/failback without shutting down are:

- ObjectServer writer
- Sybase database writer
- Sybase Reporter writer
- SNMP writer
- ServiceView writer
- Socket writer
- Flat file writer
- Informix database writer

Writers that support failover/failback with shutdown are:

- Remedy ARS writer
- Siebel eCommunications writer
- Oracle database writer
- Oracle Reporter writer
- Peregrine writer
- Clarify writer
- HP ITSM writer
- Peoplesoft Vantive writer
- HP Service Desk writer
- ODBC database writer

## 4.11 Conversion Table Utility

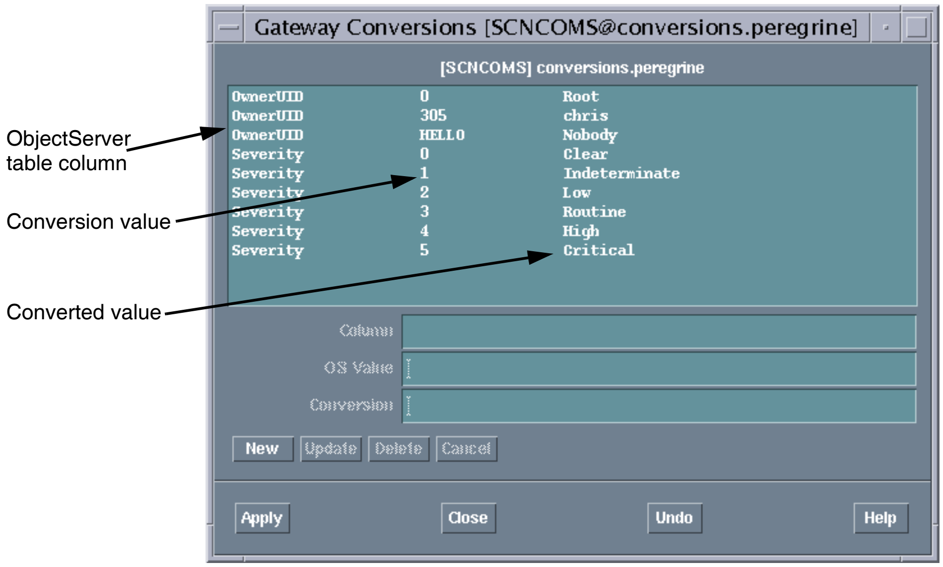
You can create conversion tables to enable certain data conversions to take place between fields. For example, in the Gateway for Peregrine, ServiceCenter users require a status field to be alphabetic and to have a particular value. The ObjectServer may hold these as numeric values.

You must also ensure that the appropriate table exists in your ObjectServer. The default table is `NCOMS.conversions.`*`targetname`*`.` In this example *`targetname`* is the name of the conversion table specific to a gateway. For example, `NCOMS.conversions.`*`peregrine`*.

To define the conversions, run the conversion utility:

```
nco_gwconv
```

The *Gateway Conversions* window for the Gateway for Peregrine is displayed in Figure 9.


Gateway Conversions [SCNCOMS@conversions.peregrine]
[SCNCOMS] conversions.peregrine
ObjectServer table column
Conversion value
Converted value
Column
OS Value
Conversion
New
Update
Delete
Cancel
Apply
Close
Undo
Help

| | | |
|---|---|---|
| OwnerUID | 0 | Root |
| OwnerUID | 305 | chris |
| OwnerUID | HELLO | Nobody |
| Severity | 0 | Clear |
| Severity | 1 | Indeterminate |
| Severity | 2 | Low |
| Severity | 3 | Routine |
| Severity | 4 | High |
| Severity | 5 | Critical |

Figure 9: Gateway For Peregrine Conversion Window

The *Gateway Conversion* window title bar contains the ObjectServer name and table name being used. The work area displays any existing conversions.

Conversion details are displayed in three columns:

- The first column contains the ObjectServer table column name.
- The second column contains the values associated with the column.
- The third column contains the converted value.

## Adding a Conversion

To add a new conversion:

1. Click the **New** button.
2. Select the **Column** field and enter the **ObjectServer** column name.
3. Enter the **ObjectServer** value in the **OS Value** field.
4. Enter the conversion value in the **Conversion** field.
5. Click **Apply**. The new conversion is added.

## Updating a Conversion

To update an existing conversion:

1. Select the conversion to update. The conversion details are populated with the existing values.
2. Update the values as required.
3. Click **Apply**.

## Deleting a Conversion

To delete a conversion:

1. Select the conversion to delete. The conversion details are populated with the existing values.
2. Click the **Delete** button. You can undo the delete by clicking the **Undo** button.
3. Click **Apply**.

# Chapter 5: Gateway Commands and Command Line Options

This chapter describes the command line options for `nco_gate`. It also describes gateway commands that are common to all gateways.

Resynchronization commands are only valid for the ObjectServer Gateway, and are therefore described in the guide for the ObjectServer Gateway.

For more information about specific gateways, refer to the documentation available for each gateway on the Micromuse Support Site.

This chapter contains the following sections:

## 5.1 Gateway Command Line Options

This section lists the command line options for `nco_gate`.

Table 19: Gateway Command Line Options (1 of 2)

| Option | Description |
|---|---|
| `-admingroup` *`string`* | Specifies the name of the UNIX user group that has administrator privileges. Members of this group can log into the gateway. The default group name is `ncoadmin`. |
| `-authenticate` `UNIX`\|`PAM`\|`HPTCB` | Specifies the authentication mode to use to verify user credentials. The options are `UNIX`, `PAM`, and `HPTCB`. The default authentication mode is `UNIX`, which means that the Posix `getpwnam` or `getspnam` function is used to verify user credentials on UNIX platforms. Depending on system setup, passwords are verified using the `/etc/password` file, the `/etc/shadow` shadow password file, NIS, or NIS+. If `PAM` is specified as the authentication mode, Pluggable Authentication Modules are used to verify user credentials. The service name used by the gateway when the PAM interface is initialized is `netcool`. PAM authentication is available on Linux, Solaris, and HP-UX 11 platforms only. If `HPTCB` is specified as the authentication mode, this HP-UX password protection system is used. This option is only available on HP trusted (secure) systems. |
| `-config` *`string`* | Specifies the name of the configuration file to be read at start up. The default is `$OMNIHOME/etc/`*`gatename`*`.conf`. |
| `-debug` | When specified, debug mode is enabled. |
| `-help` | Displays help information about the command line options and exits. |
| `-logfile` *`string`* | Specifies the name of the log file. If omitted, the default is `$OMNIHOME/log/`*`gatename`*`.log`. |
| `-logsize` *`integer`* | Specifies the maximum size of the log file in KBytes. The minimum is `16` KBytes. The default is `1` MByte. |
| `-name` *`string`* | Specifies the gateway name. Specify this name following the `-server` command line option to connect to the gateway using `nco_sql`. If omitted, the default is `NCO_GATE`. |
| `-notruncate` | Specifies that the log file is not truncated. |
| `-queue` *`integer`* | Specifies the size of the internal queues. The default is `1024`. Do not modify unless advised by Micromuse Support. |

Table 19: Gateway Command Line Options (2 of 2)

| Option | Description |
|---|---|
| `-stacksize` *`integer`* | Specifies the size of the internal threads. The default is `256` KBytes. Do not modify unless advised by Micromuse Support. |
| `-uniquelog` | If `-logfile` is not set, this option forces the log file to be uniquely named by appending the process ID of the gateway to the end of the default log file name.<br><br>If `-logfile` is set, this has no effect. |
| `-version` | Displays version information and exits. |

## 5.2 Reader Commands

This section describes the available reader commands for gateways.

## START READER

This section describes the `START READER` command.

### Syntax

The syntax of the `START READER` command is:

```
START READER reader_name CONNECT TO server_name [ USING FILTER filter_name ]
[ ORDER BY 'column, ... [ ASC | DESC ]' ] [ AFTER IDUC DO 'update_command' ]
[ IDUC = integer ] [ JOURNAL_FLUSH = integer ] [ IDUC_ORDER ];
```

### Usage

Starts a reader named *`reader_name`* which connects to an ObjectServer named *`server_name`*.

The optional `USING FILTER` clause, followed by the name of a filter that has been created using the `CREATE FILTER` command, enables you to restrict the number of rows affected by gateway updates. The filter replaces an SQL `WHERE` clause, so the gateway only updates the rows selected by the filter.

The optional `ORDER BY` clause instructs the gateway to display the results in sequential order, depending on the values of one or more column names, in either descending (`DESC`) or ascending (`ASC`) order. If the `ORDER BY` clause is not specified, no ordering is used.

The optional `AFTER IDUC` clause instructs the gateway to perform the update specified in the *`update_command`* in the ObjectServer when it places alerts in the writer queue. This is used to provide feedback when alerts pass through a gateway.

The value specified in the optional `IDUC` clause indicates an IDUC interval for gateways that is more frequent than the value of the `Granularity` property set in the source ObjectServer. This enables gateway updates to be forwarded to the target more rapidly without causing overall system performance to deteriorate.

The value specified in the optional `JOURNAL_FLUSH` clause indicates a delay in seconds between when the IDUC update occurs in the ObjectServer (every *`Granularity`* seconds) and when the journal entries are retrieved by the gateway. Normally, only journal entries that have been made in the last *`Granularity`* seconds are retrieved. When the system is under heavy load, set this clause so journal entries are retrieved for the last *`integer + Granularity`* seconds. This prevents the loss of any journal entries that are created after the IDUC update but before the gateway retrieves the entries. Any duplicate journal entries retrieved are eliminated by deduplication.

The optional IDUC_ORDER clause specifies the order in which the IDUC data is processed. The default processing mode for gateways is to process DELETE statements, followed by UPDATE statements, followed by INSERT statements. Do not change this clause unless you have been advised to do so by Micromuse Support.

### Example

```
START READER NCOMS_READER CONNECT TO NCOMS USING FILTER CRIT_ONLY
ORDER BY 'SERIAL ASC' AFTER IDUC DO 'update alerts.status set Grade=2';
```

This example uses the Grade field as a state field. Initially, all probes set Grade to 0. The gateway filters any alerts that have a Grade of 1. After the alerts have passed through the gateway, the AFTER IDUC update provides alert state feedback by changing the value of the Grade field to 2.

## STOP READER

This section describes the STOP READER command.

### Syntax

The syntax of the STOP READER command is:

```
STOP READER reader_name;
```

### Usage

Stops the reader named *reader_name*. This command will not stop the reader if the reader is in use with any routes.

### Example

```
STOP READER NCOMS_READ;
```

## SHOW READERS

This section describes the SHOW READERS command.

## Syntax

The syntax of the `SHOW READERS` command is:

```
SHOW READERS;
```

## Usage

Lists all the current readers that have been started and are running on the gateway. This command can only be used interactively.

### Example

```
SHOW READERS;
```

## 5.3 Writer Commands

This section describes the available writer commands for gateways.

## START WRITER

This section describes the `START WRITER` command.

### Syntax

The syntax of the `START WRITER` command is:

```
START WRITER writer_name
( TYPE=writer_type , REVISION=number
[ , keyword_setting [ , keyword_setting ] ...] );
```

### Usage

Starts a writer named *`writer_name`*. This is followed by a list of comma-separated keyword settings in parentheses. The first setting must be a `TYPE` setting indicating the *`writer_type`*. The next setting must be a `REVISION` setting. This is currently set to `1` for all writers. The remaining keywords and their settings depend on the type of writer.

### Example

The example shown starts the writer for a socket gateway:

```
START WRITER SOCKET_WRITER
(
        TYPE = SOCKET,
        REVISION = 1,
        HOST = 'sfo768',
        PORT = 4010,
        MAP = SOCKET_MAP,
        INSERT_HEADER = 'INSERT: ',
        UPDATE_HEADER = 'UPDATE: ',
        DELETE_HEADER = 'DELETE: ',
        START_STRING = '"',
        END_STRING = '"',
        INSERT_TRAILER = '\n',
        UPDATE_TRAILER = '\n',
        DELETE_TRAILER = '\n'
);
```

## STOP WRITER

This section describes the `STOP WRITER` command.

### Syntax

The syntax of the `STOP WRITER` command is:

```
STOP WRITER writer_name;
```

### Usage

Stops the writer called *`writer_name`*. If any route is using this writer, the writer will not be stopped.

### Example

```
STOP WRITER ARS_WRITER;
```

## SHOW WRITERS

This section describes the `SHOW WRITERS` command.

### Syntax

The syntax of the `SHOW WRITERS` command is:

```
SHOW WRITERS;
```

### Usage

Lists all current writers in the gateway. This command can only be used interactively.

### Example

```
1>SHOW WRITERS;
2>GO
Name        Type Routes Msgq Id Mutex Id Thread
----------- ---- ------ ------- -------- ------
SNMP_WRITER SNMP 1      15      0        0x001b8cd0

1>
```

## SHOW WRITER TYPES

This section describes the `SHOW WRITER TYPES` command.

### Syntax

The syntax of the `SHOW WRITER TYPES` command is:

```
SHOW WRITER TYPES;
```

### Usage

Lists all the currently known types of writers supported by the gateway. This command can only be used interactively.

### Example

```
1> SHOW WRITER TYPES;
2> GO
 Type              Revision      Description
 ------           -----------    -----------------------
 ARS              1              Action Request System V3.0
 OBJECT_SERVER    1              Netcool/OMNIbus ObjectServer V7
 SYBASE           1              Sybase SQL Server 10.0 RDBMS
 SNMP             1              SNMP Trap forwarder
 SERVICE_VIEW     1              Service View
```

## SHOW WRITER ATTRIBUTES

This section describes the `SHOW WRITER ATTRIBUTES` command.

### Syntax

The syntax of the `SHOW WRITER ATTRIBUTES` command is:

```
SHOW WRITER { ATTRIBUTES | ATTR } FOR writer_name;
```

### Usage

Shows all the settings (attributes) of the writer named *`writer_name`*. The `ATTRIBUTES` keyword is interchangeable with the abbreviated `ATTR` keyword.

This command can only be used interactively.

## Example

```
1> SHOW WRITER ATTR FOR SNMP_WRITER;
2> GO
Attribute           Value
-----------         -------------------------------------------------------
MAP                 SNMP_MAP
TYPE                SNMP
REVISION            1
GATEWAY             penelope

1>
```

## 5.4 Mapping Commands

This section describes the available mapping commands for gateways.

### CREATE MAPPING

This section describes the `CREATE MAPPING` command.

#### Syntax

The syntax of the `CREATE MAPPING` command is:

```
CREATE MAPPING mapping_name ( mapping [ , mapping ] );
```

#### Usage

Creates a mapping file named *mapping_name* for use by a writer. Mapping lines have the following syntax:

```
{ string | integer } = { string | integer | name | real | boolean }
[ ON INSERT ONLY ] [ CONVERT TO { INT | STRING | DATE } ]
```

The first argument is an identifier for the destination field and the second argument is an identifier for the source field (or a preset value).

The right-hand side of the mapping is dependent on the writer with which the mapping is to be used. Refer to the appropriate writer section of the individual gateway guide, available on the Micromuse Support Site.

The optional `ON INSERT ONLY` clause determines the update behavior of the mapping. Without the `ON INSERT ONLY` clause, the field is updated every time a change is made to an alert. With the `ON INSERT ONLY` clause, the field is inserted at creation time (that is, when the alert appears for the first time) but is not updated on subsequent updates of the alert even if the field value is changed.

The optional `CONVERT TO` type clause allows the mapping to define a forced conversion for situations where a source field may not match the type of the destination field. The type can be `INT`, `STRING`, or `DATE`. This forces the source field to be converted to the specified data type.

#### Example

```
CREATE MAPPING SYBASE_MAP
(
'Node'='@Node' ON INSERT ONLY,
'Summary'='@Summary' ON INSERT ONLY,
'Severity'='@Severity' );
```

## DROP MAPPING

This section describes the `DROP MAPPING` command.

### Syntax

The syntax of the `DROP MAPPING` command is:

```
DROP MAPPING mapping_name;
```

### Usage

Removes the mapping named *mapping_name* from the gateway. This command will not drop the map if it is being used by a writer.

### Example

```
DROP MAPPING SYBASE_MAP;
```

## SHOW MAPPINGS

This section describes the `SHOW MAPPINGS` command.

### Syntax

The syntax of the `SHOW MAPPINGS` command is:

```
SHOW MAPPINGS;
```

### Usage

Lists all the mappings currently created in the gateway. This command can only be used interactively.

### Example

```
1> SHOW MAPPINGS;
2> GO
 Name                             Writers
 -------------------------------- -----------
 SNMP_MAP                                   1
1>
```

## SHOW MAPPING ATTRIBUTES

This section describes the `SHOW MAPPING ATTRIBUTES` command.

### Syntax

The syntax of the `SHOW MAPPING ATTRIBUTES` command is:

```
SHOW MAPPING { ATTRIBUTES | ATTR } FOR mapping_name;
```

### Usage

Shows the mappings (attributes) of the mapping named *mapping_name*. The `ATTRIBUTES` keyword is interchangeable with the abbreviated `ATTR` keyword. This command can only be used interactively.

### Example

```
SHOW MAPPING ATTR FOR SYBASE_MAP;
```

## 5.5 Filter Commands

This section describes the available filter commands for gateways.

## CREATE FILTER

This section describes the `CREATE FILTER` command.

### Syntax

The syntax of the `CREATE FILTER` command is:

```
CREATE FILTER filter_name AS filter_condition;
```

### Usage

Creates a filter named *filter_name* for use by a reader. The filter specification *filter_condition* is an SQL condition. SQL conditions are described in the Netcool/OMNIbus Administration Guide.

#### Example

```
CREATE FILTER HIGH_TALLY_LOG AS 'Tally > 100';
CREATE FILTER NCOMS_FILTER AS 'Agent = \'NNM\'';
```

## LOAD FILTER

This section describes the `LOAD FILTER` command.

### Syntax

The syntax of the `LOAD FILTER` command is:

```
LOAD FILTER FROM 'filename';
```

### Usage

Loads a filter from a file. Filter files have the `.elf` file extension.

#### Example

```
LOAD FILTER FROM '/disk/filters/newfilter.elf';
```

## DROP FILTER

This section describes the `DROP FILTER` command.

### Syntax

The syntax of the `DROP FILTER` command is:

```
DROP FILTER filter_name;
```

### Usage

Removes the filter named *`filter_name`* from the gateway. The filter will not be dropped if it is being used by a reader.

#### Example

```
DROP FILTER HIGH_TALLY_LOG;
```

## 5.6 Route Commands

This section describes the available route commands for gateways.

## ADD ROUTE

This section describes the `ADD ROUTE` command.

### Syntax

The syntax of the `ADD ROUTE` command is:

```
ADD ROUTE FROM reader_name TO writer_name;
```

### Usage

Adds a route between a reader named *`reader_name`* and a writer named *`writer_name`* to allow alerts to pass through the gateway.

#### Example

```
ADD ROUTE FROM NCOMS_READER TO ARS_WRITER;
```

## REMOVE ROUTE

This section describes the `REMOVE ROUTE` command.

### Syntax

The syntax of the `REMOVE ROUTE` command is:

```
REMOVE ROUTE FROM reader_name TO writer_name;
```

### Usage

Removes an existing route between a reader named *`reader_name`* and a writer named *`writer_name`*.

#### Example

```
REMOVE ROUTE FROM NCOMS_READER TO ARS_WRITER;
```

## SHOW ROUTES

This section describes the SHOW ROUTES command.

### Syntax

The syntax of the SHOW ROUTES command is:

```
SHOW ROUTES;
```

### Usage

Shows all currently configured routes in the gateway. This command can only be used interactively.

### Example

```
1> SHOW ROUTES;
2> GO
 Reader                             Writer
 --------------------------------- ---------------------------------
 NCOMS_READER                       SNMP_WRITER

1>
```

## 5.7 Configuration Commands

This section describes the available configuration commands for gateways.

## LOAD CONFIG

This section describes the `LOAD CONFIG` command.

### Syntax

The syntax of the `LOAD CONFIG` command is:

```
LOAD CONFIG FROM 'filename';
```

### Usage

Loads a gateway configuration file from a file named in *`filename`*.

#### Example

```
LOAD CONFIG FROM '/disk/config/gateconf.conf';
```

## SAVE CONFIG

This section describes the `SAVE CONFIG` command.

### Syntax

The syntax of the `SAVE CONFIG` command is:

```
SAVE CONFIG TO 'filename';
```

### Usage

Saves the current configuration of the gateway into a file named in *`filename`*.

#### Example

```
SAVE CONFIG TO '/disk/config/newgate.conf';
```

## DUMP CONFIG

This section describes the `DUMP CONFIG` command.

## Syntax

The syntax of the DUMP CONFIG command is:

```
DUMP CONFIG [ FORCE ];
```

## Usage

Clears the current configuration. If the gateway is active and forwarding alerts, this command will not clear the configuration unless the optional keyword FORCE is used.

### Example

```
DUMP CONFIG;
```

## 5.8 General Commands

This section describes the available general commands for gateways.

### SHUTDOWN

This section describes the `SHUTDOWN` command.

#### Syntax

The syntax of the `SHUTDOWN` command is:

```
SHUTDOWN [ FORCE ];
```

#### Usage

Instructs the gateway to shut down; all readers and writers are stopped. By default, the gateway is not shut down if interactive changes to the configuration have not been saved. Refer to *SHOW SYSTEM* on page 115 for details on how to determine if the configuration has been changed interactively.

If the optional `FORCE` keyword is used, the gateway is shut down, even if the configuration has been changed interactively.

#### Example

```
SHUTDOWN;
```

### SET CONNECTIONS

This section describes the `SET CONNECTIONS` command.

#### Syntax

The syntax of the `SET CONNECTIONS` command is:

```
SET CONNECTIONS { TRUE | FALSE | YES | NO };
```

#### Usage

Enables or disables connections to the gateway using the SQL interactive interface. When set to `FALSE` or `NO`, it is not possible to connect to the gateway with `nco_sql`. When set to `TRUE` or `YES`, it is possible to connect to the gateway with `nco_sql`. This command determines whether interactive reconfiguration is allowed.

### Example

```
SET CONNECTIONS TRUE;
```

## SHOW SYSTEM

This section describes the `SHOW SYSTEM` command.

### Syntax

The syntax of the `SHOW SYSTEM` command is:

```
SHOW SYSTEM;
```

### Usage

Displays information about the current gateway settings. The parameters returned are shown in Table 20.

Table 20: Show System Parameters

| System Parameter | Description |
|---|---|
| Version | Version number of the gateway. |
| Server Type | Type of server. Should be `Gateway`. |
| Connections | Status of the `SET CONNECTIONS` flag. Refer to *SET CONNECTIONS* on page 114. |
| Debug Mode | Status of the `SET DEBUG MODE` flag. Refer to *SET DEBUG MODE* on page 116. |
| Multi User | Gateway multi-user mode. Should be `YES`. |
| Configuration Changed | If the configuration has been changed interactively, this is set to `YES`. |

More parameters may be returned when in debug mode. This command can only be used interactively.

### Example

```
1> SHOW SYSTEM;
2> GO
System Parameter     Value
----------------     --------------------------------
 Version             3.6
 Server Type         Gateway
 Connections         ENABLED
 Debug Mode          NO
 Multi User          YES
```

## SET DEBUG MODE

This section describes the `SET DEBUG MODE` command.

### Syntax

The syntax of the `SET DEBUG MODE` command is:

```
SET DEBUG MODE { TRUE | FALSE | YES | NO };
```

### Usage

Sets the debugging mode of the gateway. When set to `TRUE` or `YES`, debugging messages are sent to the log file. The default setting is `NO` or `FALSE`. This command should only be used under the advice of Micromuse Support.

### Example

```
SET DEBUG MODE NO;
```

## TRANSFER

This section describes the `TRANSFER` command.

### Syntax

The syntax of the `TRANSFER` command is:

```
TRANSFER 'tablename' FROM readername TO writername [ AS 'tableformat' ]
{ DELETE | DELETE condition | DO NOT DELETE }
[ USE TRANSFER_MAP ] [ USING FILTER filter_clause ];
```

### Usage

Transfers the contents of one database table to another database table. You can use this command to transfer tables between Sybase, Oracle, Informix, ODBC, CORBA, and Socket gateways.

The `AS` *`tableformat`* clause specifies the format of the destination table if it is different from the source table format.

The DELETE and DO NOT DELETE clauses define how the destination table is processed. By default, the contents of the destination table are deleted before the contents of the source table are transferred. You can optionally specify a condition that determines whether the deletion will occur. If you specify DO NOT DELETE, the contents of the destination table are not deleted before the contents of the source table are transferred.

**Note:** The DELETE clause does not function with the Socket and CORBA gateways.

The USE TRANSFER_MAP clause instructs the gateway to use the mapping definition that is assigned as the map to the writer used in the TRANSFER command. The USE TRANSFER_MAP clause is only available for use with the Oracle Gateway.

An optional filter clause may be applied by specifying USING FILTER followed by the filter. Enter a valid filter, as described in the CREATE FILTER command.

## Example

```
TRANSFER 'alerts.conversions' FROM NCO_READER TO SYBASE_WRITER AS
'alerts.conversions' DELETE;
TRANSFER 'alerts.status' FROM NCOMS_READ TO DENCO_WRITE AS 'ncoms.status'
USING FILTER 'ServerName = \'NCOMS\'' DELETE USE TRANSFER_MAP;
```

# Appendix A: Regular Expressions

This appendix contains information about how to use regular expressions. It contains the following section:

## A.1 How to Use Regular Expressions

Regular expressions are made up of normal characters and metacharacters. Normal characters include upper and lower case letters and numbers. Regular expression pattern matching can be performed with either a single character or a pattern of one or more characters within parentheses, called a *character pattern*. Metacharacters have special meanings, described in Table A1.

Table A1: Pattern Matching Metacharacters (1 of 2)

| Pattern Matching Metacharacter | Description | Example |
|---|---|---|
| * | Matches zero or more instances of the preceding character or character pattern. | The pattern 'goo*' matches 'my godness', 'my goodness', and 'my gooodness', but not 'my gdness'. |
| + | Matches one or more instances of the preceding character or character pattern. | The pattern 'goo+' matches 'my goodness' and 'my gooodness', but not 'my godness'. |
| ? | Matches zero or one instance of the preceding character or character pattern. | The pattern 'goo?' matches 'my godness' and 'my goodness', but not 'my gooodness' or 'my gdness'. |
| $ | Matches the end of the string. | The pattern 'end$' matches 'the end', but not 'the ending'. |
| ^ | Matches the beginning of the string. | The pattern '^severity' matches 'severity level 5', but not 'The severity is 5'. |
| . | Matches any single character. | The pattern 'b.at' matches 'baat', 'bBat', and 'b4at', but not 'bat' or 'bB4at'. |
| [abcd] | Matches any characters in the square brackets or in the range of characters separated by a hyphen (-), such as [0-9]. | ^[A-Za-z]+$ matches any string that contains only upper or lower case letter characters. |
| [^abcd] | Matches any character except those in the square brackets or in the range of characters separated by a hyphen (-), such as [0-9]. | [^0-9] matches any string that does not contain any numeric characters. |
| () | Indicates that the characters within the parentheses should be treated as a character pattern. | A(boo)+Z matches 'AbooZ', 'AboobooZ', and 'AbooboobooZ', but not 'AboZ' or 'AboooZ'. |
| \| | Matches one of the characters or character patterns on either side of the vertical bar. | A(B\|C)D matches 'ABD' and 'ACD', but not 'AD', 'ABCD', 'ABBD', or 'ACCD'. |

Table A1: Pattern Matching Metacharacters (2 of 2)

| Pattern Matching Metacharacter | Description | Example |
|---|---|---|
| \ | The backslash escape character indicates that the metacharacter following should be treated as a regular character. The metacharacters in this table require a backslash before them if they appear in a regular expression. | To match an opening square bracket, followed by any digits or spaces, followed by a closed bracket, use the regular expression \[[0-9 ]*\]. |

# Appendix B: ObjectServer Tables

This appendix contains ObjectServer database table information. It contains the following sections:

- *Alerts Tables* on page 124
- *Service Tables* on page 133
- *ObjectServer Data Types* on page 134

## B.1 Alerts Tables

Alert information is forwarded to the ObjectServer from external programs such as probes, TSMs, and monitors, stored and managed in database tables, and displayed in the event list.

### alerts.status Table

The `alerts.status` table contains status information about problems that have been detected by probes, TSMs, and monitors.

Table B1: Columns in the alerts.status Table (1 of 7)

| Column Name | Data Type | Mandatory | Description |
|---|---|---|---|
| `Identifier` | `varchar(255)` | Yes | Controls ObjectServer deduplication. |
| `Serial` | `incr` | Yes | The Netcool/OMNIbus serial number for the row. |
| `Node` | `varchar(64)` | Yes | Identifies the managed entity from which the alarm originated. This could be a host or device name, service name, or other entity. For IP network devices or hosts, the `Node` column contains the resolved name of the device or host. In cases where the name cannot be resolved, the `Node` column should contain the IP address of the host or device. |
| `NodeAlias` | `varchar(64)` | No | The alias for the node. For network devices or hosts, this should be the logical (layer-3) address of the entity. For IP devices or hosts, this should be the IP address. |
| `Manager` | `varchar(64)` | Yes | The descriptive name of the probe that collected and forwarded the alarm to the ObjectServer. This can also be used to indicate the host on which the probe is running. |
| `Agent` | `varchar(64)` | No | The descriptive name of the sub-manager that generated the alert. |
| `AlertGroup` | `varchar(64)` | No | The descriptive name of the type of failure indicated by the alert (for example, `Interface Status` or `CPU Utilization`). |
| `AlertKey` | `varchar(64)` | Yes | The descriptive key that indicates the managed object instance referenced by the alert (for example, the disk partition indicated by a file system full alert or the switch port indicated by a utilization alert). |

Table B1: Columns in the alerts.status Table (2 of 7)

| Column Name | Data Type | Mandatory | Description |
|---|---|---|---|
| `Severity` | `integer` | Yes | Indicates the alert severity level, which provides an indication of how the perceived capability of the managed object has been affected. The color of the alert in the event list is controlled by the severity value:<br>`0` - Clear<br>`1` - Indeterminate<br>`2` - Warning<br>`3` - Minor<br>`4` - Major<br>`5` - Critical |
| `Summary` | `varchar(255)` | Yes | The text summary of the cause of the alert. |
| `StateChange` | `time` | Yes | An automatically maintained ObjectServer timestamp of the last insert or update of the alert from any source. |
| `FirstOccurrence` | `time` | Yes | The time in seconds (from midnight Jan 1, 1970) when this alert was created or when polling started at the probe. |
| `LastOccurrence` | `time` | Yes | The time when this alert was last updated at the probe. |
| `InternalLast` | `time` | Yes | The time when this alert was last updated at the ObjectServer. |
| `Poll` | `integer` | No | The frequency of polling for this alert in seconds. |

Table B1: Columns in the alerts.status Table (3 of 7)

| Column Name | Data Type | Mandatory | Description |
|---|---|---|---|
| `Type` | `integer` | No | The type of alert:<br>`0` - Type not set<br>`1` - Problem<br>`2` - Resolution<br>`3` - Netcool/Visionary problem<br>`4` - Netcool/Visionary resolution<br>`7` - Netcool/ISMs new alarm<br>`8` - Netcool/ISMs old alarm<br>`11` - More Severe<br>`12` - Less Severe<br>`13`- Information |
| `Tally` | `integer` | Yes | Automatically maintained count of the number of inserts and updates of the alert from any source. |
| `Class` | `integer` | Yes | The alert class used to identify the probe or vendor from which the alert was generated. Controls the applicability of context-sensitive event list tools. |
| `Grade` | `integer` | No | Indicates the state of escalation for the alert:<br>`0` - Not Escalated<br>`1` - Escalated |
| `Location` | `varchar(64)` | No | Indicates the physical location of the device, host, or service for which the alert was generated. |
| `OwnerUID` | `integer` | Yes | The user identifier of the user who is assigned to handle this alert.<br>The default is `65534`, which is the identifier for the `nobody` user. |
| `OwnerGID` | `integer` | No | The group identifier of the group that is assigned to handle this alert.<br>The default is `0`, which is the identifier for the `public` group. |

Table B1: Columns in the alerts.status Table (4 of 7)

| Column Name | Data Type | Mandatory | Description |
|---|---|---|---|
| `Acknowledged` | `integer` | Yes | Indicates whether the alert has been acknowledged:<br>`0` - No<br>`1` - Yes<br>Alerts can be acknowledged manually by a network operator or automatically by a correlation or workflow process. |
| `Flash` | `integer` | No | Enables the option to make the event list flash. |
| `EventID` | `varchar(64)` | No | The event ID (for example, `SNMPTRAP-link down`). Multiple events can have the same event ID. This is populated by the probe rules file and used by Netcool/Precision. |
| `ExpireTime` | `integer` | Yes | The number of seconds until the alert is cleared automatically. Used by the Expire automation. |
| `ProcessReq` | `integer` | No | Indicates whether the alert should be processed by Netcool/Precision. This is populated by the probe rules file and used by Netcool/Precision. |
| `SuppressEscl` | `integer` | Yes | Used to suppress or escalate the alert:<br>`0` - Normal<br>`1` - Escalated<br>`2` - Escalated-Level 2<br>`3` - Escalated-Level 3<br>`4` - Suppressed<br>`5` - Hidden<br>`6` - Maintenance<br>The suppression level is manually selected by operators from the event list. |
| `Customer` | `varchar(64)` | No | The name of the customer affected by this alert. |
| `Service` | `varchar(64)` | No | The name of the service affected by this alert. |
| `PhysicalSlot` | `integer` | No | The slot number indicated by the alert. |

Table B1: Columns in the alerts.status Table (5 of 7)

| Column Name | Data Type | Mandatory | Description |
|---|---|---|---|
| `PhysicalPort` | `integer` | No | The port number indicated by the alert. |
| `PhysicalCard` | `varchar(64)` | No | The card name or description indicated by the alert. |
| `TaskList` | `integer` | Yes | Indicates whether a user has added the alert to the Task List:<br>`0` - No<br>`1` - Yes<br>Operators can add alerts to the Task List from the event list. |
| `NmosSerial` | `varchar(64)` | No | The serial number of a suppressed alert. Populated by Netcool/Precision. |
| `NmosObjInst` | `integer` | No | Populated by Netcool/Precision during alert processing. |
| `NmosCauseType` | `integer` | No | The alert state, populated by Netcool/Precision as an integer value:<br>`0` - Unknown<br>`1` - Root cause<br>`2` - Symptom |
| `LocalNodeAlias` | `varchar(64)` | Yes | The alias of the network entity indicated by the alert. For network devices or hosts, this is the logical (layer-3) address of the entity, or another logical address that enables direct communication with the device. For use in managed object instance identification. |
| `LocalPriObj` | `varchar(255)` | No | The primary object referenced by the alert. For use in managed object instance identification. |
| `LocalSecObj` | `varchar(255)` | No | The secondary object referenced by the alert. For use in managed object instance identification. |
| `LocalRootObj` | `varchar(255)` | Yes | An object that is equivalent to the primary object referenced in the alarm. For use in managed object instance identification. |
| `RemoteNodeAlias` | `varchar(64)` | Yes | The network address of the remote network entity. For use in managed object instance identification. |

Table B1: Columns in the alerts.status Table (6 of 7)

| Column Name | Data Type | Mandatory | Description |
|---|---|---|---|
| `RemotePriObj` | `varchar(255)` | No | The primary object of a remote network entity referenced by an alarm. For use in managed object instance identification. |
| `RemoteSecObj` | `varchar(255)` | No | The secondary object of a remote network entity referenced by an alarm. For use in managed object instance identification. |
| `RemoteRootObj` | `varchar(255)` | Yes | An object that is equivalent to the remote entity's primary object referenced in the alarm. For use in managed object instance identification. |
| `X733EventType` | `integer` | No | Indicates the alert type:<br>`0` - Not defined<br>`1` - Communications<br>`2` - Quality of Service<br>`3` - Processing error<br>`4` - Equipment<br>`5` - Environmental<br>`6` - Integrity violation<br>`7` - Operational violation<br>`8` - Physical violation<br>`9` - Security service violation<br>`10` - Time domain violation |
| `X733ProbableCause` | `integer` | No | Indicates the probable cause of the alert. |
| `X733SpecificProb` | `varchar(64)` | No | Identifies additional information for the probable cause of the alert. Used by probe rules files to specify a set of identifiers for use in managed object instance identification. |
| `X733CorrNotif` | `varchar(255)` | No | A listing of all notifications with which this notification is correlated. |
| `ServerName` | `varchar(64)` | Yes | The name of the originating ObjectServer. Used by gateways to control propagation of alerts between ObjectServers. |

Table B1: Columns in the alerts.status Table (7 of 7)

| Column Name | Data Type | Mandatory | Description |
|---|---|---|---|
| `ServerSerial` | `integer` | Yes | The serial number of the alert on the originating ObjectServer (if it did not originate on this ObjectServer). Used by gateways to control the propagation of alerts between ObjectServers. |
| `URL` | `varchar(1024)` | No | Optional URL which provides a link to additional information in the vendor's device or ENMS. |
| `MasterSerial` | `integer` | No | Identifies the master ObjectServer if this alert is being processed in a desktop ObjectServer environment.<br><br>This column is added when you run `nco_dbinit` with the `-desktopserver` option.<br><br>**Note:** `MasterSerial` must be the last column in the `alerts.status` table if you are using a desktop ObjectServer environment.<br><br>For information about the desktop ObjectServer environment, see the Netcool/OMNIbus Installation and Deployment Guide. |

**Note:** You can only display columns of type `CHAR`, `VARCHAR`, `INCR`, `INTEGER`, and `TIME` in the event list. Do not add columns of any other type to the `alerts.status` table.

## alerts.details Table

The `alerts.details` table contains the detail attributes of the alerts in the system.

Table B2: Columns in the alerts.details Table (1 of 2)

| Column Name | Data Type | Description |
|---|---|---|
| `KeyField` | `varchar(255)` | Internal sequencing string for uniqueness. |
| `Identifier` | `varchar(255)` | Identifier to relate details to entries in the `alerts.status` table. |
| `AttrVal` | `integer` | Boolean; when false (`0`), just the `Detail` column is valid. Otherwise, the `Name` and `Detail` columns are both valid. |
| `Sequence` | `integer` | Sequence number, used for ordering entries in the event list *Event Information* window. |

Table B2: Columns in the alerts.details Table (2 of 2)

| Column Name | Data Type | Description |
|---|---|---|
| `Name` | `varchar(255)` | Name of attribute stored in `Detail` column. |
| `Detail` | `varchar(255)` | Attribute value. |

## alerts.journal Table

The `alerts.journal` table provides a history of work performed on alerts.

Table B3: Columns in the alerts.journal Table (1 of 2)

| Column Name | Data Type | Description |
|---|---|---|
| `KeyField` | `varchar(255)` | Primary key for table. |
| `Serial` | `integer` | Serial number of alert that this journal entry is related to. |
| `UID` | `integer` | User identifier of user who made this entry. |
| `Chrono` | `time` | Time and date that this entry was made. |
| `Text1` | `varchar(255)` | First block of text for journal entry. |
| `Text2` | `varchar(255)` | Second block of text for journal entry. |
| `Text3` | `varchar(255)` | Third block of text for journal entry. |
| `Text4` | `varchar(255)` | Fourth block of text for journal entry. |
| `Text5` | `varchar(255)` | Fifth block of text for journal entry. |
| `Text6` | `varchar(255)` | Sixth block of text for journal entry. |
| `Text7` | `varchar(255)` | Seventh block of text for journal entry. |
| `Text8` | `varchar(255)` | Eighth block of text for journal entry. |
| `Text9` | `varchar(255)` | Ninth block of text for journal entry. |
| `Text10` | `varchar(255)` | Tenth block of text for journal entry. |

Table B3: Columns in the alerts.journal Table (2 of 2)

| Column Name | Data Type | Description |
|---|---|---|
| `Text11` | `varchar(255)` | Eleventh block of text for journal entry. |
| `Text12` | `varchar(255)` | Twelfth block of text for journal entry. |
| `Text13` | `varchar(255)` | Thirteenth block of text for journal entry. |
| `Text14` | `varchar(255)` | Fourteenth block of text for journal entry. |
| `Text15` | `varchar(255)` | Fifteenth block of text for journal entry. |
| `Text16` | `varchar(255)` | Sixteenth block of text for journal entry. |

## B.2 Service Tables

The service table contains information about Netcool/ISMs.

### service.status Table

The `service.status` table is used to control the additional features required to support Netcool/ISMs.

Table B4: Columns in the service.status Table

| Column Name | Data Type | Description |
|---|---|---|
| `Name` | `varchar(255)` | Name of the service. |
| `CurrentState` | `integer` | Indicates the state of the service:<br>`0` - Good<br>`1` - Bad<br>`2` - Marginal<br>`3` - Unknown |
| `StateChange` | `time` | Indicates the last time the service state changed. |
| `LastGoodAt` | `time` | Indicates the last time the service was Good (`0`). |
| `LastBadAt` | `time` | Indicates the last time the service was Bad (`1`). |
| `LastMarginalAt` | `time` | Indicates the last time the service was Marginal (`2`). |
| `LastReportAt` | `time` | Time of the last service status report. |

## B.3 ObjectServer Data Types

Each column value in the ObjectServer has an associated data type. The data type determines how the ObjectServer processes the data in the column. For example, the plus operator (+) adds integer values or concatenates string values, but does not act on boolean values. The data types supported by the ObjectServer are listed in Table B5:

Table B5: ObjectServer Data Types

| SQL Type | Description | Default Value | ObjectServer ID for Data Type |
|---|---|---|---|
| `INTEGER` | 32 bit signed integer. | `0` | `0` |
| `INCR` | 32 bit unsigned auto-incrementing integer. Applies to table columns only, and can only be updated by the system. | `0` | `5` |
| `UNSIGNED` | 32 bit unsigned integer. | `0` | `12` |
| `BOOLEAN` | `TRUE` or `FALSE.` | `FALSE` | `13` |
| `REAL` | 64 bit signed floating point number. | `0.0` | `14` |
| `TIME` | Time, stored as the number of seconds since midnight January 1, 1970. This is the Coordinated Universal Time (UTC) international time standard. | `Thu Jan 1 01:00:00 1970` | `1` |
| `CHAR(`*`integer`*`)` | Fixed size character string, *`integer`* characters long (8192 Bytes is the maximum).<br><br>The `char` type is identical in operation to `varchar`, but performance is better for mass updates that change the length of the string. | `''` | `10` |
| `VARCHAR(`*`integer`*`)` | Variable size character string, up to *`integer`* characters long (8192 Bytes is the maximum).<br><br>The `varchar` type uses less storage space than the `char` type and the performance is better for deduplicatation, scanning, insert, and delete operations. | `''` | `2` |
| `INTEGER64` | 64 bit signed integer. | `0` | `16` |
| `UNSIGNED64` | 64 bit unsigned integer. | `0` | `17` |

**Note:** You can only display columns of type `CHAR`, `VARCHAR`, `INCR`, `INTEGER`, and `TIME` in the event list. Do not add columns of any other type to the `alerts.status` table.

# Appendix C: Probe Error Messages and Troubleshooting Techniques

This appendix lists all of the messages that are common to all probes, including ProbeWatch and TSMWatch messages.

Refer to the individual probe guides for information about probe-specific messages.

This appendix also includes troubleshooting information for probes.

This appendix contains the following sections:

## C.1 Generic Error Messages

Probes can generate the following types of messages:

- Fatal
- Error
- Warning
- Information
- Debug

### Fatal Level Messages

The probe automatically terminates when a fatal message is issued.

Table C1: Fatal Level Probe Messages (1 of 2)

| Message | Description | Action |
|---|---|---|
| `Connection to ObjectServer marked DEAD - aborting...` | The connection to the ObjectServer ceased (and store and forward is not enabled in the probe). | Check that the ObjectServer is available. |
| `Failed to access OMNIHOME directory: "directory name"`<br><br>`Failed to set interfaces file location` | The probe was unable to locate the interfaces file. | Check that the `OMNIHOME` environment variable is set to the correct destination. |
| `Failed to connect - aborting` | The ObjectServer is not available. | Check that the ObjectServer is running, that the interfaces file on the system where the probe is installed has an entry for the ObjectServer, and that there is no networking problem between the two systems. |

Table C1: Fatal Level Probe Messages (2 of 2)

| Message | Description | Action |
|---|---|---|
| `Failed to create property`<br>`Failed to define argument`<br>`Failed to initialise`<br>`Failed to set property`<br>`Failed to process arguments`<br>`Session create failed - aborting` | Internal errors. | Refer to your support contract for information about contacting the helpdesk. |
| `Failed to read rules - aborting` | A property or command line option is pointing to a non-existent rules file. | Check that the command line option or properties file refers to the correct rules file. |
| `Field "field name" not found in status table`<br>`No matching field found for "field name"` | The rules file being used refers to a field of the format `@fieldname` which does not exist in the status table. | Check the rules file and correct the problem. |
| `Unknown data type returned from ObjectServer` | The ObjectServer has returned unknown data. | Refer to your support contract for information about contacting the helpdesk. |

## Error Level Messages

The probe is likely to terminate when an error message is issued.

Table C2: Error Level Probe Messages (1 of 4)

| Message | Description | Action |
|---|---|---|
| `Can't set generic property "property name" via command line`<br>`Property "property name" for option "option name" does not exist` | An option in the probe is not mapped correctly to a property. | Check the properties file for the named property and refer to the probe documentation for supported properties. |
| `Could not send alert` | The probe was unable to send an alert (usually an internal alert) to the ObjectServer. | Check that the ObjectServer is available. |

Table C2: Error Level Probe Messages (2 of 4)

| Message | Description | Action |
|---|---|---|
| `Could not set "fieldname" field` | The probe was unable to set a field value. This may be because the ObjectServer tables have been modified so that default fields are no longer present. | Check if the ObjectServer tables have been modified. |
| `CreateAndSet failed`<br><br>`CreateAndSet failed for attr: "element name"` | The probe is unable to create an element. | Refer to your support contract for information about contacting the helpdesk. |
| `Error Setting SIGINT Handler`<br><br>`Error Setting SIGQUIT Handler`<br><br>`Error Setting SIGTERM Handler` | The probe was unable to set up a signal handler for either an `INT`, `QUIT`, or `TERM`. | Refer to your support contract for information about contacting the helpdesk. |
| `Failed to open file: "file name"` | A file referred to in the rules file (for example, with the `table` function) does not exist. | Check the rules file and ensure the file is available. |
| `Failed to open message log: "file name"` | The probe is unable to open the specified log file. | Check the command line or properties file and correct the problem. |
| `Failed to open Properties file: "properties file name"` | The probe is unable to open the properties file. | Check the properties file or command line to ensure the properties file is in the specified location. |
| `Failed to open Rules file: "rules file name"`<br><br>`The rules file for the probe is not available or incorrectly specified.` | The probe is unable to open the rules file. | Check the properties file or command line to ensure the rules file is in the specified location. |
| `No extraction data for "regexp" - missing ()'s?`<br><br>`Regexp doesn't match for "string"` | A regular expression being used in the `extract` function may be missing parentheses.<br><br>The string data that is being used to extract may not match the regular expression.<br><br>The `extract` function is unable to extract data. | Check the rules file and correct the problem. |

Table C2: Error Level Probe Messages (3 of 4)

| Message | Description | Action |
|---|---|---|
| `Option "option name" used without argument` | The option used expects a value which has not been supplied. | Check the probe documentation and the contents of the command line. |
| `OS Error: "error message"`<br><br>`Procedure "procedure name": "error message"`<br><br>`Server "server name": "error message"` | There is an error in the Sybase connection. There should be a subsequent message from the probe which details the effect of this error. | Refer to your support contract for information about contacting the helpdesk. |
| `Properties file: "error description" at line "line no"` | There is an error in the format of the properties file. | Check the properties file at the specified line number and correct the problem. |
| `PropGetValue failed` | A required property has not been set. | Check the properties file. |
| `Regular Expression Error: "regexp"` | A regular expression is incorrectly formed in the rules file. | Check the rules file for the regular expression and correct the problem. |
| `Results processing failed`<br><br>`Unexpected return from results processing`<br><br>`Unexpected value during results processing` | There is a problem with the ObjectServer. | Refer to your support contract for information about contacting the helpdesk. |
| `Rules file: "error description" at line "line no"` | There is an error in the rules file format or syntax. | Check the rules file at the specified line number and correct the problem. |
| `SendAlert failed` | The probe was unable to send an alert to the ObjectServer. | Check that the ObjectServer is available. |
| `SessionProcess failed` | The probe was unable to process the alert against the rules file. | Refer to your support contract for information about contacting the helpdesk. |
| `Unknown message level "message level string" - using WARNING level` | The properties file or command line specified a message level which is not supported. | Check the properties file or command line and use a supported message level (`debug`, `info`, `warning`, `error`, `fatal`). |
| `Unknown option: "option name"` | An option has been used on the command line to start the probe which is not supported by the probe. | Check the probe documentation and the contents of the command line. |

Table C2: Error Level Probe Messages (4 of 4)

| Message | Description | Action |
|---|---|---|
| `Unknown property "property name" - ignored` | A property specified in the properties file does not exist in the probe. | Check the properties file for the named property and refer to the probe documentation for supported properties. |

## Warning Level Messages

These messages are issued as warnings but should not cause the probe to terminate.

Table C3: Warning Level Probe Messages

| Message | Description | Action |
|---|---|---|
| `Failed to install Client Message Callback`<br>`Failed to install Server Message Callback`<br>`Failed to retrieve connection status - attempting to continue`<br>`Results processing failed` | There is a problem with the ObjectServer. | The probe will try to continue. |
| `Failed to set SYBASE in environment` | The probe was unable to override the `SYBASE` environment variable. | Check that the `SYBASE` environment variable is correctly set. |
| `New value for field "field name" truncated to "number" characters` | A string being copied into an alert field has had to be truncated to fit the field. | Check the rules file. |
| `Type mismatch for property "property name" - new value ignored` | A property has been set with the wrong data type. | Check the properties file or command line to ensure that the property is correctly set. |

## Information Level Messages

This message is for information purposes.

Table C4: Information Level Probe Messages

| Message | Description | Action |
|---|---|---|
| `Using stderr for logging` | The probe was unable to open a log file. | No action required. The probe is writing messages to `stderr`. |

## Debug Level Messages

Debug level messages provide information about the internal functions of the probe. These messages are aimed at probe developers but are listed here for completeness.

Table C5: Debug Level Probe Messages (1 of 3)

| Message | Description | Action |
|---|---|---|
| `A value for "string" doesn't exist in lookup table "table name"` | A value requested from a lookup table is not available. | No action required. The function in the rules file will return an empty string. |
| `Attempted to duplicate NULL string`<br><br>`Attempted to free NULL pointer`<br><br>`Attempted to realloc NULL pointer`<br><br>`Failed to allocate memory (Requested size was "number" bytes)`<br><br>`Failed to duplicate string`<br><br>`Failed to reallocate memory block at address "hex address" (Requested size was "number" bytes)` | An error or problem has occurred in the memory allocation or string handling components of the probe library. | No action required. The library will handle the problem. |

Table C5: Debug Level Probe Messages (2 of 3)

| Message | Description | Action |
|---|---|---|
| `Failed to allocate command structure`<br>`Failed to allocate context structure`<br>`Failed to bind column`<br>`Failed to connect`<br>`Failed to describe column`<br>`Failed to fetch number of columns`<br>`Failed to initialise Sybase internals: "number"`<br>`Failed to send command`<br>`Failed to set appname`<br>`Failed to set command query`<br>`Failed to set hostname`<br>`Failed to set password`<br>`Failed to set username`<br>`Got a row fail - continuing`<br>`No columns in result set` | A problem or error has occurred at the Sybase or ObjectServer connection level. | N/A |
| `Failed to flush alerts before EXIT`<br>`Problem during disconnect before EXIT`<br>`Problem during session destruction before EXIT`<br>`Problem during shutdown before EXIT` | A problem has occurred during probe shutdown. | N/A |
| `New value for field "field name" is "value"` | A field value has been set. | N/A |

Table C5: Debug Level Probe Messages (3 of 3)

| Message | Description | Action |
|---|---|---|
| `OplInitialise() called more than once` | Multiple calls have been made to the `OplInitialise` C probe API function , which can only be called once. | N/A |

## C.2 ProbeWatch and TSMWatch Messages

In some situations, a probe or TSM generates events of its own. These events can provide information (such as startup or shutdown messages) or identify problems. This section describes the elements common to all ProbeWatch and TSMWatch messages.

ProbeWatch and TSMWatch messages are processed in the rules file and converted into alerts like other events. Table C6 shows the elements common to ProbeWatch and TSMWatch events.

Table C6: Common ProbeWatch and TSMWatch Elements

| Element Name | Description |
|---|---|
| `Summary` | Summary string, described in the following tables. |
| `Node` | Name of the node on which the probe or TSM is running. |
| `Agent` | Name of the probe or TSM. |
| `Manager` | `ProbeWatch` or `TSMWatch`. |

Table C7 describes summary strings common to all probes and TSMs.

Table C7: Common ProbeWatch and TSMWatch Summary Strings

| ProbeWatch/TSMWatch Message | Description | Cause |
|---|---|---|
| `Going down ...` | The probe or TSM is shutting down. | The probe or TSM is executing a shutdown routine. |
| `Running ...` | The probe or TSM has started running. | The probe or TSM has just been started. |
| `Unable to get events ...` | The probe or TSM encountered a problem while listening for events. | There was a problem initializing the connection or there was a license or connection failure after some events were received. Refer to your support contract for information about contacting the helpdesk. |

Refer to the individual probe guides for additional summary strings for each probe.

TSMWatch messages are in the same format as ProbeWatch messages. Table C8 describes summary strings common to all TSMs.

Table C8: Common TSMWatch Summary Strings

| TSMWatch Message | Description | Action |
|---|---|---|
| `Connection Attempted ...`<br>`Connection Succeeded ...`<br>`Connection Failed ...`<br>`Connection Timed out ...`<br>`Connection Lost ...` | Messages relating to the establishment of a TCP/IP connection. | N/A |
| `Disconnection Attempted ...`<br>`Disconnection Succeeded ...`<br>`Disconnection Failed ...` | Messages relating to relinquishing a TCP/IP connection. | N/A |
| `Wakeup Attempted ...`<br>`Wakeup Succeeded ...`<br>`Wakeup Failed ...` | Messages relating to wake up functionality. | N/A |
| `Login Attempted ...`<br>`Login Succeeded ...`<br>`Login Timed out ...`<br>`Login Failed ...` | Messages relating to host login. | N/A |
| `Logout Attempted ...`<br>`Logout Succeeded ...`<br>`Logout Timed out ...`<br>`Logout Failed ...` | Messages relating to host logout. | N/A |
| `Heartbeat Sent ...`<br>`Heartbeat Received ...`<br>`Heartbeat Timed out ...` | Messages relating to sending/receiving heartbeat messages to/from the host. | N/A |
| `Resynchronisation Attempted ...`<br>`Resynchronisation Succeeded ...`<br>`Resynchronisation Failed ...` | Messages relating to synchronizing current alerts between the switch and Netcool/OMNIbus. | N/A |

## C.3 Troubleshooting Probes

This section describes some of the common problems experienced by Netcool/OMNIbus users and explains possible causes and solutions.

This troubleshooting information is divided into two sections:

- Common problem causes
- What to do if

Table C9: Troubleshooting Probes

| Section | Description |
|---|---|
| *Common Problem Causes* on page 148 | This section contains a list of common problem causes. If you are unsure what your problem is, you should start by reading this part and following the instructions. If you cannot solve your problem by following the instructions in this part, move on to the section *What to Do If* on page 149. |
| *What to Do If* on page 149 | This section describes common symptoms caused by probe problems and step-by-step instructions to help you locate and solve the problem. If none of the headings in this section match the symptoms of your problem, read through the lists of instructions and make sure that you have tried all of the most likely solutions listed there. |

## Common Problem Causes

The most common causes of probe problems are:

- Incorrectly set `OMNIHOME` and `NETCOOL_LICENSE_FILE` environment variables
- Errors in the rules file, particularly in `extract` statements
- Configuration errors in the properties file

For information about setting the `OMNIHOME` and `NETCOOL_LICENSE_FILE` environment variables, refer to the Netcool/OMNIbus Installation and Deployment Guide.

For information about solving rules file problems, refer to Chapter 2: *Probe Rules File Syntax* on page 27.

For information about probe properties, refer to Chapter 3: *Probe Properties and Command Line Options* on page 59. Check that all of the properties are set correctly in the probe properties file. For example, check that the `Server` property contains the correct ObjectServer or proxy server name and that the `RulesFile` property contains the correct rules file name.

If you cannot solve the problem, read through the next section and make sure that you have tried all of the most likely solutions listed there.

## What to Do If

The headings in this section describe the most common symptoms of probe problems. Find the heading that most closely describes your problem and follow the instructions until you have located the cause and solved the problem:

Table C10: Types of Probe Problems

| Problem | See... |
|---|---|
| The probe does not start. | page 149 |
| The probe is not sending alerts to the ObjectServer. | page 151 |
| The probe is losing events. | page 152 |
| The probe is consuming too much CPU time. | page 152 |
| The event list fields are not being populated properly. | page 153 |

If none of the headings match the symptoms of your problem, read through the lists of instructions and make sure that you have tried all of the most likely solutions listed there. If you have tried all of the suggested problem solutions and your probe still does not work, refer to your support contract and contact the helpdesk.

### The Probe Does Not Start

If the probe does not start:

1. Run the probe in debug mode as described in *Debugging Rules Files* on page 55.
2. Check that the ObjectServer is running by trying to connect using `nco_ping` or `nco_sql`.

   If you can connect successfully, the ObjectServer is running. If the ObjectServer is not running, this is likely to be the cause of the problem. For more information about using the ObjectServer and `nco_sql`, refer to the Netcool/OMNIbus Administration Guide.
3. Check that there are no other probes running with the same configuration using the commands:

   ```
   ps -ef | grep nco_p
   ```

   A list of probe processes is displayed. Check that none of the processes correspond to the same type of probe. You cannot run two identical probe configurations because this would duplicate all of the events forwarded to the ObjectServer.
4. Check that you have enough licenses available to start another probe by entering:

   ```
   $NCLICENSE/bin/nc_print_license
   ```

If you do not have enough licences to run another probe and you cannot stop any of the other probes, contact the helpdesk to request another license.

5. Check that you are using the correct probe for the current version of the target software.
6. Check that there are no syntax errors in the rules file. Refer to *Testing Rules Files* on page 54 for more information about how to do this.
7. Check that your system has not run out of system resources and can launch more processes. You can do this using `df -k` or `top`. Refer to the `df` and `top` man pages for more information about using these commands.
8. Check to see if the `$OMNIHOME/var/`*`probename`*`.saf` store and forward file exists. If it exists, check that it has not become too large. If your disk is full, the probes and ObjectServers are not able to work properly.

**Warning:** Store and forward is not designed to handle very large numbers of events. Left unattended, a store and forward file will continue to grow until it runs out of disk space. Refer to *Probe Properties and Command Line Options* on page 59 for information about setting the `MaxSAFFileSize` property.

9. Check that the store and forward file has not been corrupted. If the store and forward file has been corrupted there should be an error message in the log file (`$OMNIHOME/log/`*`probename`*`.log`). If the file is corrupted, delete it and restart the probe.
10. Check that the probe binary you are trying to run is the correct one for the current architecture by entering:

    ```
    $OMNIHOME/bin/arch/probename -version
    ```

    Check that the probe version matches your system architecture.

    *If you are running the probe on a remote host:*
11. Check that the probe host can connect to the ObjectServer host using the `ping` command. Try to ping the ObjectServer host machine using the hostname and the IP address. Refer to the `ping` man page for more information about how to do this.

    If you cannot connect to the ObjectServer host using the `ping` command, there is a problem with the connection between your probe host and your ObjectServer host.
12. Check that the ObjectServer has been configured correctly in the Server Editor (`nco_xigen`) and that the interfaces information has been distributed to the ObjectServer and probe hosts. Refer to the Netcool/OMNIbus Installation and Deployment Guide for more information.
13. Check to see if there is a firewall between the probe host and the ObjectServer host. If there is, make sure that the firewall will allow traffic between the probe and the ObjectServer.

## The Probe Is Not Sending Alerts to the ObjectServer

If the probe is not sending alerts to the ObjectServer:

1. Check that the probe is running by entering:

   ```
   ps -ef | grep nco_p
   ```

   A list of probe processes is displayed. If the probe is not running, start the probe from the command line.

2. Check that there are no other probes running with the same configuration by entering:

   ```
   ps -ef | grep nco_p
   ```

   A list of probe processes is displayed. Check that none of the processes correspond to the same type of probe. You cannot run two identical probe configurations because this would duplicate all of the events forwarded to the ObjectServer.

3. Read the probe properties file and check that all of the properties are set correctly. For example, check that the `Server` property contains the correct ObjectServer name and that the `RulesFile` property contains the correct rules file name.

4. Check that the probe event source has events to send to the ObjectServer.

5. Check that the ObjectServer you are logged in to is the same ObjectServer that the probe is forwarding events to.

6. Check that the event source you are trying to probe is working correctly. Refer to the documentation supplied with your element manager for more information about how to do this.

7. Check that you are using the correct probe.

8. Check that the probe is not running in store and forward mode. To do this, check the `$OMNIHOME/var/`*`probename`*`.saf` and `$OMNIHOME/var/`*`probename`*`.reco` files to see if they are growing. If they are, disable store and forward mode. Refer to *Store and Forward Mode* on page 19 for more information.

9. Check that your system has not run out of system resources and can launch more processes. You can do this using `df -k` or `top`. Refer to the `df` and `top` man pages for more information about using this command.

10. Check for any `discard` functions in the probe rules file. The `discard` function must be in a conditional statement; otherwise, all events are discarded. Refer to *Deleting Elements or Events* on page 40 for more information.

    *If you are running the probe on a remote host:*

11. Check that the probe host can connect to the ObjectServer host using the `ping` command. Try to ping the ObjectServer host machine using the hostname and the IP address. Refer to the `ping` man page for more information about how to do this.

    If you cannot connect to the ObjectServer host using the `ping` command, there is a problem with the connection between your probe host and your ObjectServer host.

12. Check that the ObjectServer has been configured correctly through the Server Editor (`nco_xigen`) and that the interfaces information has been distributed to the ObjectServer and probe hosts. Refer to the Netcool/OMNIbus Installation and Deployment Guide for more information.

13. Check to see if there is a firewall between the probe host and the ObjectServer host. If there is, make sure that the firewall will allow traffic between the probe and the ObjectServer.

## The Probe Is Losing Events

If not all of the events are being forwarded to the ObjectServer:

1. Run the probe in debug mode as described in *Debugging Rules Files* on page 55.
2. Check that the event source you are trying to probe is working correctly. Refer to the documentation supplied with your element manager for more information about how to do this.
3. Check that the probe event source has events to send to the ObjectServer.
4. Check that all of the properties in the properties file are set correctly. For example, check that the `Server` property contains the correct ObjectServer name and that the `RulesFile` property contains the correct rules file name.
5. Check for any `discard` functions in the probe rules file. The `discard` function discards events based on specified conditions. Refer to *Deleting Elements or Events* on page 40 for more information.

## The Probe Is Consuming Too Much CPU Time

If the probe is consuming too much CPU time:

1. Run the probe in debug mode as described in *Debugging Rules Files* on page 55.
2. Check that the probe can connect to the event source.

3. Check to see if the `$OMNIHOME/var/`*`probename`*`.saf` store and forward file exists. If it exists, check that it has not become too large. If your disk is full, the probes and ObjectServer will not be able to work properly.

**Warning:** Store and forward is not designed to handle very large numbers of alerts. Left unattended, a store and forward file will continue to grow until it runs out of disk space. Refer to *Probe Properties and Command Line Options* on page 59 for information about setting the `MaxSAFFileSize` property.

4. Check that the store and forward file has not been corrupted. If the store and forward file has been corrupted there should be an error message in the probe log file (`$OMNIHOME/log/`*`probename`*`.log`). If the store and forward file is corrupted, delete it and restart the probe.

## The Event List Is Not Being Populated Properly

If the probe is detecting events and forwarding them to the ObjectServer but the event list fields are not being populated correctly:

1. Run the probe in debug mode as described in *Debugging Rules Files* on page 55.
2. Check that fields which are not being populated properly are being correctly mapped to elements in the rules file. Refer to Chapter 2: *Probe Rules File Syntax* on page 27 for more information about configuring rules files.
3. Check that it is not a GUI problem by querying the `alerts.status` table using ObjectServer SQL. Refer to the Netcool/OMNIbus Administration Guide for more information about using the SQL interactive interface to view the table information.

# Appendix D: Gateway Error Messages

This appendix lists gateway error messages. It contains the section:

## D.1 Common Gateway Error Messages

This section describes error messages that can occur in all gateways. The *`gateway_name`* in each error message refers to the individual gateway name and indicates which gateway generated the error.

Table D1: Common Gateway Error Messages (1 of 12)

| Error | Description | Action |
|---|---|---|
| *`Gateway_name`* `Writer: HashAlloc failure in _` *`gateway_name`* `CacheAdd().`<br><br>*`Gateway_name`* `Writer: MemStrDup() failure in _` *`gateway_name`* `CacheAdd().` | The gateway failed to allocate memory. | Try to free more memory. |
| *`Gateway_name`* `Writer: Failed to allocate memory.`<br><br>*`Gateway_name`* `Writer` *`writer_name`*`: Memory allocation failed.`<br><br>*`Gateway_name`* `Writer: Memory allocation failure.`<br><br>*`Gateway_name`* `Writer: Memory allocation error.`<br><br>*`Gateway_name`* `Writer: Memory reallocation error.`<br><br>`Failed to allocate memory in writer` *`writer_name`*`.` | The gateway failed to allocate memory. | Try to free more memory. |
| *`Gateway_name`* `Writer` *`writer_name`*`: Could not create serial cache - memory problems.`<br><br>*`Gateway_name`* `Writer` *`writer_name`*`: Failed to allocate memory for a GPCModule handle.` | The gateway failed to allocate memory. | Try to free more memory. |

Table D1: Common Gateway Error Messages (2 of 12)

| Error | Description | Action |
|---|---|---|
| `*Gateway_name*` `Writer: Failed to lock connection mutex.` | The writer failed to lock the ObjectServer feedback connection in order to access the connection and feed back problem ticket data for the associated alert. This lock failure may be due to insufficient resources or as a result of the underlying threading system preventing a deadlock between multiple threads that are contending for the resource. | Refer to your support contract for information about contacting the helpdesk. |
| *Gateway_name* `Writer: Failed to re-acquire alert details from OS.` | This error message comes from the gateway cache reclamation sub-system. This message indicates that the gateway failed to re-acquire the trouble ticket number and reclaim its internal cache entry from the ObjectServer. | Refer to your support contract for information about contacting the helpdesk. |
| *Gateway_name* `Writer: Invalid datatype for problem number feedback field.` | The data type is invalid. | Refer to the Netcool/OMNIbus Administration Guide for information about data types. |
| *Gateway_name* `Writer: Serial x already in serial Cache. Cannot add.` | The gateway tried to add a serial number that already exists. | Refer to your support contract for information about contacting the helpdesk. |
| *Gateway_name* `Writer: Serial x not found in serial cache. Cannot Delete.` | The gateway could not delete this alert because it has already been deleted in Netcool/OMNIbus. | You do not need to do anything. |
| *Gateway_name* `Writer` *writer_name*`: Failed to construct path to` *gateway_name* `Read/Write Module.` | The gateway could not locate the reader/writer module application. | Check that the module is installed in the correct location. |
| *Gateway_name* `Writer` *writer_name*`: Failed to construct the argument list for` *gateway_name* `Module.` | Failed to construct the argument list for gateway module. | Check that the arguments in the configuration file are set correctly. |
| *Gateway_name* `Writer` *writer_name*`: GPCModule creation failed.` | Failed to create the GPCModule due to insufficient memory. | Try to free more memory. |

Table D1: Common Gateway Error Messages (3 of 12)

| Error | Description | Action |
|---|---|---|
| `Gateway_name Writer writer_name: Failed to start the OS-gateway_name Writer.`<br><br>`Gateway_name Writer writer_name: Failed to start the gateway_name-OS Reader.` | Failed to start the ObjectServer to gateway reader or writer module. | Check that the module is installed in the correct location and that the file permissions are set correctly. |
| `Gateway_name Writer writer_name: Failed to shutdown gateway_name Writer.` | Failed to stop gateway writer module. | Check the writer log file for more information. |
| `Gateway_name Writer writer_name: Failed to construct path to gateway_name Read/Write Module.` | Failed to construct the path to the gateway read/write module application. | Check that the module is installed in the correct location and that the file permissions are set correctly. |
| `Gateway_name Writer writer_name: Failed to find the gateway_name Read/Write Module [x].` | Cannot find the module binary. | Check that the module is installed in the correct location and that the file permissions are set correctly. |
| `Gateway_name Writer writer_name: Incorrect permissions on the gateway_name module binary [x].` | The module's file permissions are set incorrectly. | Check that the module is installed in the correct location and that the file permissions are set correctly. |
| `Gateway_name Writer writer_name: Failed to create the Serial Cache Mutex.` | The gateway writer failed to create the necessary data protection structure for the internal serial number cache due to insufficient resources. This is generally due to insufficient memory. | Try to free more memory. |
| `Gateway_name Writer writer_name: Failed to create the Conn Mutex.` | The gateway writer failed to create the necessary data protection structure for the ObjectServer connection due to insufficient resources. | Try to free more memory. |
| `Gateway_name Writer writer_name: Failed to start the gateway_name-to-OS service thread.` | The gateway failed to spawn the service thread. | Check that the gateway can access the ObjectServer. |

Table D1: Common Gateway Error Messages (4 of 12)

| Error | Description | Action |
|---|---|---|
| `*Gateway_name* Writer *writer_name*: Failed to send a shutdown request to the *gateway_name* Writer.` | The gateway did not shut down cleanly. | Check the writer log file for more information. |
| `Failed to install SIGCHLD handler.`<br>`Failed to install SIGPIPE handler.` | The gateway failed during handler installation. | Refer to your support contract for information about contacting the helpdesk. |
| `No <mapname> attribute for *gateway_name* writer *writer_name*.` | The gateway could not find the map name. | Check the configuration file. |
| `<mapname> attribute is not a name for *gateway_name* writer *writer_name*.` | Incorrect writer name given. | Check the configuration file. |
| `A MAP called <map> does not exist for *gateway_name* writer *writer_name*.` | The gateway could not find specified map. | Check the configuration file. |
| `MAP <map> is invalid for *gateway_name* writer *writer_name*.` | The given map is not valid. | Check the configuration file. |
| `Map <map> is not the journal map and cannot contain the <journal map name> map item in *gateway_name* Writer *writer_name*.` | If this map is not the journal map, then the `JOURNAL_MAP_NAME` attribute is set incorrectly. | Check the `JOURNAL_MAP_NAME` attribute in the gateway configuration file. |
| `*Gateway_name* Writer: Failed to send *gateway_name* Event to the *gateway_name* Writer module.` | The gateway failed to send a given event. | Check the log files for more information. |
| `*Gateway_name* Writer: Failed to wait for return from the *gateway_name* Writer module.` | There was an error in retrieving the success statement. | Check the log files for more information. |
| `*Gateway_name* Writer: Failed to read the status return message from the *gateway_name* Writer module.` | The gateway failed to retrieve the status of a module. | Check the log files for more information. |

Table D1: Common Gateway Error Messages (5 of 12)

| Error | Description | Action |
|---|---|---|
| `Gateway_name Writer: Failed to send event to gateway_name.` | The module failed to send the event to gateway. | Check the log files for more information. |
| `Gateway_name Writer: gateway_name Writer Module experienced Fatal Error.` | There was a fatal error. | Check the log files for more information. |
| `Gateway_name Writer: Failed to send event to gateway_name. Unknown type.` | The gateway received unexpected type. | Refer to your support contract for information about contacting the helpdesk. |
| `Gateway_name Writer: Failed to build serial index.` | The gateway failed to build indexes. | Check that the `Serial` column exists in the ObjectServer `alerts.status` table. |
| `Incorrect data type for the Serial column.` | The gateway did not receive the correct data type. | Check that the data type for the `Serial` column in the ObjectServer `alerts.status` table is an integer. |
| `Gateway_name Writer: Failed to build server serial index.` | The gateway failed to get the server serial index. | Check that the `ServerSerial` column exists in the ObjectServer `alerts.status` table. |
| `Incorrect data type for the Server Serial column.` | The gateway did not receive the correct data type. | Check that the data type for the `ServerSerial` column in the ObjectServer `alerts.status` table is an integer. |
| `Gateway_name Writer: Failed to build server name index.` | The gateway failed to get the server name index. | Check that the `ServerName` column exists in the ObjectServer `alerts.status` table. |
| `Incorrect data type for the Server Name column.` | The gateway did not receive the correct data type. | Check that the data type for the `ServerName` column in the ObjectServer `alerts.status` table is a string. |
| `Gateway_name Writer: Failed to find field <fieldnumber> in gateway_name Event.` | The gateway could not find the field number it was looking for. | Refer to your support contract for information about contacting the helpdesk. |

Table D1: Common Gateway Error Messages (6 of 12)

| Error | Description | Action |
|---|---|---|
| `*Gateway_name*` `Writer: Invalid field name for expansion on action SQL [<field>].` | The gateway received an invalid field name. | Refer to the Netcool/OMNIbus Administration Guide for information about ObjectServer SQL. |
| `*Gateway_name*` `Writer: Unenclosed field expansion request in action SQL [<sql action>].` | The gateway did not find an enclosing bracket. | Check the `action.sql` file. |
| *Gateway_name* `Writer: Failed to turn counter-part notification back-on. Fatal error in` *gateway_name*`-to-OS Feedback.`<br><br>*Gateway_name* `Writer: Failed to turn counter-part notification off.`<br><br>*Gateway_name*`-to-OS Feedback failed.` | The gateway failed to send a notify command. | This is an internal error. Refer to your support contract for information about contacting the helpdesk. |
| *Gateway_name* `Writer: Failed to send SQL command to ObjectServer.`<br><br>*Gateway_name*`-to-OS Feedback failed.` | The gateway failed to send the SQL command to the ObjectServer. | Check the ObjectServer log file. |
| `Failed to find the column <column_name> in map <map_name>.` | The gateway failed to find the given column. | Check that the given column name is entered correctly in the configuration file and that it appears in the ObjectServer `alerts.status` table. |
| *Gateway_name* `Writer: Failed to lock the cache mutex.` | The writer failed to lock the ObjectServer feedback connection in order to access the connection and feed back problem ticket data changes for the associated alert. | This lock failure may be due to insufficient resources or as a result of the underlying threading system preventing a deadlock between multiple threads that are contending for the resource. |
| `Failed to find cached problem ticket for serial <serial number> using map <map name>.` | The gateway failed to find the specified cache problem ticket number. | Check that the specified ticket was originally created by the gateway. |

Table D1: Common Gateway Error Messages (7 of 12)

| Error | Description | Action |
|---|---|---|
| `*Gateway_name* Writer: Failed to unlock the cache mutex.` | After access to the cache, an attempt to unlock the data structures protection lock failed. This message indicates that the gateway is in a position which will lead to a deadlock situation. | Refer to your support contract for information about contacting the helpdesk. |
| `*Gateway_name* Writer: Cache add error.` | The gateway could not add the serial to the serial cache due to insufficient resources. | Try to free more memory. |
| `*Gateway_name* Writer *writer_name*: Failed to create *gateway_name* Event for journal update.` | The gateway failed to create the journal event update. | Check the writer log file. |
| `*Gateway_name* Writer *writer_name*: Failed to send journal update event to *gateway_name*.` | The gateway failed to send journal event update. | Check the writer log file. |
| `<attribute name> attribute is not a string for *gateway_name* writer *writer_name*.` | An attribute in the writer was of an incorrect data type. | Check the writer definition in the configuration file. |
| `No <attribute name> attribute for *gateway_name* writer *writer_name* given.` | The gateway failed to find the attribute. | Add the attribute to the writer definition in the configuration file. |
| `*Gateway_name* Writer *writer_name*: Failed to find the <counterpart attribute> attribute for the writer. This is necessary due to bi-directional nature.` | An attempt to find the necessary counterpart attribute failed. | Check the configuration file. |
| `*Gateway_name* Writer *writer_name*: Is not a name for an Object Server reader.` | The gateway found an incorrect data type. | Check the configuration file. |
| `*Gateway_name* Writer *writer_name*: Reader <reader> was not found for counter part.` | The reader was not found. | Check the counterpart configuration in the configuration file. |

Table D1: Common Gateway Error Messages (8 of 12)

| Error | Description | Action |
|---|---|---|
| `*Gateway_name* Writer *writer_name*: Failed to send SKIP Command.` | This command failed to disable IDUC on a bidirectional connection. | Refer to your support contract for information about contacting the helpdesk. |
| `Connection to feedback server failed.` | The gateway failed to make connection. | Check the ObjectServer log file. |
| `Failed to set the death call on the feedback connection.` | The gateway failed to set the necessary property. | This is an internal error. Refer to your support contract for information about contacting the helpdesk. |
| `Writer counterpart error.` | The gateway failed to find the counterpart attribute for gateway writer. | Check the counterpart configuration in the configuration file. |
| `*Gateway_name* Writer: Failed to stat() the action SQL file "*filename*".` | The gateway failed to `stat` the file in order to determine its size. | Check the file access permissions for the specified action file. |
| `*Gateway_name* Writer: Empty action SQL file "*filename*".` | File "*`filename`*" is empty. | Check the action SQL file. |
| `*Gateway_name* Writer: Failed to open the action SQL file "*filename*".` | The gateway failed to open the file. | Check the file permissions. |
| `*Gateway_name* Writer: Failed to read the action SQL file "*filename*".` | The gateway failed to read the file. | Check the file permissions. |
| `*Gateway_name* Writer: No Action SQL find in file "*filename*".` | There is no action SQL in the file. | Check the file. |
| `*Gateway_name* Writer *writer_name*: Failed to read the conversions table.` | The gateway failed to read the conversions table. | Check the file permissions. |

Table D1: Common Gateway Error Messages (9 of 12)

| Error | Description | Action |
|---|---|---|
| `*Gateway_name* Writer: Failed to find PM %s in cache for return PMO event.` | The gateway has received a Problem Management Open return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache, in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found. | Refer to your support contract for information about contacting the helpdesk. |
| `*Gateway_name* Writer: Open Feedback Failed.` | The gateway failed to construct the open action SQL statement or send the SQL action command to the server. | Check the ObjectServer SQL file. |
| `*Gateway_name* Writer: No Update action SQL for *gateway_name* Update event.` | There is no update action SQL statement. | Check the configuration file. |
| `*Gateway_name* Writer: Failed to find PM %s in cache for return PMU event.` | The gateway has received a Problem Management Update return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found. | Refer to your support contract for information about contacting the helpdesk. |
| `*Gateway_name* Writer: Update Feedback Failed.` | The gateway failed to construct the open action SQL statement or send the SQL action command to the server. | Check the ObjectServer log file. |
| `*Gateway_name* Writer: Failed to find PM %s in cache for return PMJ event.` | The gateway has received a Problem Management Journal return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found. | Refer to your support contract for information about contacting the helpdesk. |

Table D1: Common Gateway Error Messages (10 of 12)

| Error | Description | Action |
|---|---|---|
| `Gateway_name Writer: Journal Feedback Failed.` | The gateway failed to construct the open action SQL statement or send the SQL action command to the server. | Check the ObjectServer log file. |
| `Gateway_name Writer: Failed to find PM %s in cache for return PMC event.` | The gateway has received a Problem Management Close return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found. | Refer to your support contract for information about contacting the helpdesk. |
| `Gateway_name Writer: Close Feedback Failed.` | The gateway failed to construct the open action SQL statement or send the SQL action command to the server. | Check the ObjectServer log file. |
| `Received error code <code> from Reader/Writer Module - [<message>].` | The gateway received an error message. | Check the module log files. |
| `Gateway_name Writer: Failed to read gateway_name event from gateway_name Reader Module.` | The gateway failed to read the event sent by the gateway reader module. | Check the reader log files. |
| `Gateway_name Writer: Received event of type <event type> which was unexpected.` | The gateway received an unknown event type. | Refer to your support contract for information about contacting the helpdesk. |
| `Gateway_name Writer: Received invalid known message from Reader/Writer Module for this system.` | The gateway received an invalid known message. | Refer to your support contract for information about contacting the helpdesk. |
| `Gateway_name Writer: Received unknown message from Reader/Writer Module.` | The gateway received an invalid unknown message. | Refer to your support contract for information about contacting the helpdesk. |

Table D1: Common Gateway Error Messages (11 of 12)

| Error | Description | Action |
|---|---|---|
| `Gateway_name Writer: Failed to block on data feed from gateway_name Reader Module.` | The gateway failed to block due to a shutdown request. This message is displayed when the gateway is shutting down. | Refer to your support contract for information about contacting the helpdesk. |
| `Gateway_name Writer: Fatal thread termination. Stopping gateway.` | A thread exited unexpectedly. | Check the gateway log files. |
| `<attribute name> attribute is not a string for gateway_name writer writer_name - IGNORED` | An attribute name is not recognized. The gateway will ignore it. | Check the gateway log files. |
| `<attribute name> attribute must be set to TRUE or FALSE for writer writer_name.` | An attribute name has not been set to `TRUE` or `FALSE`. | Check the gateway configuration file. |
| `Gateway_name Writer writer_name: Failed to shutdown gateway_name Reader/Writer Modules.` | The gateway failed to shut down reader/writer modules. | Check the module log file. |
| `Gateway_name Writer writer_name: Failed to disconnect feedback connection.` | The disconnect of feedback channel failed. | Check the ObjectServer log file. |
| `Failed to create gateway_name event structure for a problem management open event in writer writer_name.` | The gateway writer failed to allocate a gateway event structure for a problem management open event due to insufficient memory resources. | Try to free more memory. |
| `Gateway_name Writer: FEEDBACK FAILED!!` | The gateway failed to store the problem number. | Check the ObjectServer log file. |
| `Failed to create journal for gateway_name writer writer_name (from INSERT)` | The gateway failed to create journal. | Check the writer log file. |
| `Failed to create gateway_name event structure for a problem management update event in writer writer_name.` | The gateway writer failed to allocate a gateway event structure for a problem management update event due to insufficient memory resources. | Try to free more memory. |

Table D1: Common Gateway Error Messages (12 of 12)

| Error | Description | Action |
|---|---|---|
| `Gateway_name Writer writer_name: Failed to delete problem ticket from cache for serial <serial number>.` | The gateway failed to delete serial number from cache. | This is an internal error. You can ignore it. |
| `Failed to create gateway_name event structure for a PMC event in writer writer_name.` | The gateway writer failed to allocate a gateway event structure for a Problem Management Close event due to insufficient memory resources. | Try to free more memory. |

# Index

# Contact Information

Corporate

| Region | Address | Telephone | Fax | World Wide Web |
|---|---|---|---|---|
| **USA** | Micromuse Inc. (HQ)<br>139 Townsend Street<br>San Francisco<br>CA 94107<br>USA | 1-800-Netcool (638 2665)<br><br>+1 415 538 9090 | +1 415 538 9091 | http://www.micromuse.com |
| **EUROPE** | Micromuse Ltd.<br>Disraeli House<br>90 Putney Bridge Road<br>London SW18 1DA<br>United Kingdom | +44 (0) 20 8875 9500 | +44 (0) 20 8875 9995 | http://www.micromuse.co.uk |
| **ASIA-PACIFIC** | Micromuse Ltd.<br>Level 2<br>26 Colin Street<br>West Perth<br>Perth WA 6005<br>Australia | +61 (0) 8 9213 3400 | +61 (0) 8 9486 1116 | http://www.micromuse.com.au |

Technical Support

| Region | Telephone | Fax |
|---|---|---|
| **USA** | 1-800-Netcool (800 638 2665)<br><br>+1 415 538 9090 (San Francisco) | +1 415 538 9091 |
| **EUROPE** | +44 (0) 20 8877 0073 (London, UK) | +44 (0) 20 8875 0991 |
| **ASIA-PACIFIC** | +61 (0) 8 9213 3470 (Perth, Australia) | +61 (0) 8 9486 1116 |
| | E-mail | World Wide Web |
| **GLOBAL** | support@micromuse.com | http://support.micromuse.com |

License Generation Team

| E-Mail | World Wide Web |
|---|---|
| licensing@micromuse.com | http://support.micromuse.com/helpdesk/licenses |