Preside Multiservice Data Manager

Application Programming Interface

Primer

241-6001-200

Preside Multiservice Data Manager

Application Programming Interface

Primer

Publication: 241-6001-200
Document status: Standard
Document version: 14.3RSUP
Document date: December 2003

Publication history

December 2003

14.3 RSUP Standard
Commercial availability

Contents

About this document

The following topics are discussed in this section:

Who should read this document and why

This document is for customers and programmers who want an overview of Preside Multiservice Data Manager (MDM) Application Programming Interfaces (API).

What you need to know

To use any of the APIs described in this document

- You must be familiar with the UNIX operating system.
- You must be familiar with a programming tool or language such as C-shell scripts.
- You must have access to an Preside Multiservice Data Manager (MDM) workstation, or be able to remotely access a workstation.

How this document is organized

241-6001-200 *Preside MDM Application Programming Interface Primer* contains the following sections:

- "Introducing MDM APIs" (page 15) explains what an API is, and outlines the fundamental concepts of using APIs.
- "Current APIs" (page 19) explains what types of data you can access with each API, and gives you guidelines on the API to use.
- "Managed object model" (page 23) outlines the object model used in APIs.
- "API messages" (page 41) lists the different types of API messages, and shows you how to generate them and how to interpret them.
- "Sieve object class definitions" (page 63) provides the definition of the sieve object class and its attributes.
- "Connecting your custom program to an API" (page 71) explains how to link your custom program to an API.
- "Message syntax summary" (page 79) provides a set of tables that summarize the syntax and procedures of Preside Multiservice Data Manager (MDM) APIs.

What's new in this document

There are no changes in this document for this release.

Text conventions

This document uses the following text conventions:

- `nonproportional spaced plain type`

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

- **nonproportional spaced bold type**

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

- *italics*

  Statements that appear in italics in a procedure explain the results of a particular step and appear immediately following the step.

  Words that appear in italics in text are for naming.

- `[optional_parameter]`

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

- `<general_term>`

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

- UPPERCASE,lowercase

  In Preside Multiservice Data Manager (MDM), uppercase and lowercase letters that appear in UNIX commands and parameters must be matched exactly. The system matches upper and lowercase characters differently.

- |

  This symbol separates items from which you may select one; for example, ON|OFF indicates that you may specify ON or OFF. If you do not make a choice, a default of ON is assumed.

- ...

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

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

Related documents

See the following documents for related information:

- 241-6001-118 *Preside MDM SNMP Surveillance Adapter Guide*
- 241-6001-201 *Preside MDM Network Model API Reference Guide*
- 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*
- 241-6001-204 *Preside MDM DPN Provisioning API Reference Guide*
- 241-6001-303 *Preside MDM Administrator Guide*

Chapter 1 Introducing MDM APIs

This section introduces Preside Multiservice Data Manager (MDM) Application Programming Interfaces (APIs). This section contains the following information:

What are MDM APIs

Preside Multiservice Data Manager (MDM) Application Programming Interfaces (APIs) have the following characteristics:

- open, published interfaces
- allow other network management systems and custom programs external to the MDM application to access the full range of network data
- access to network data includes any data that the network may contain about devices that are not supported by MDM
- used by software produced by Nortel Networks, such as MDM workstation software

In the request-response model used by APIs, the external application issues requests to the API, which responds appropriately.

Each API lets you retrieve a specific type of information. See “Current APIs” (page 19) for information on the available APIs, and the APIs to use for different types of data.

With APIs, external applications can

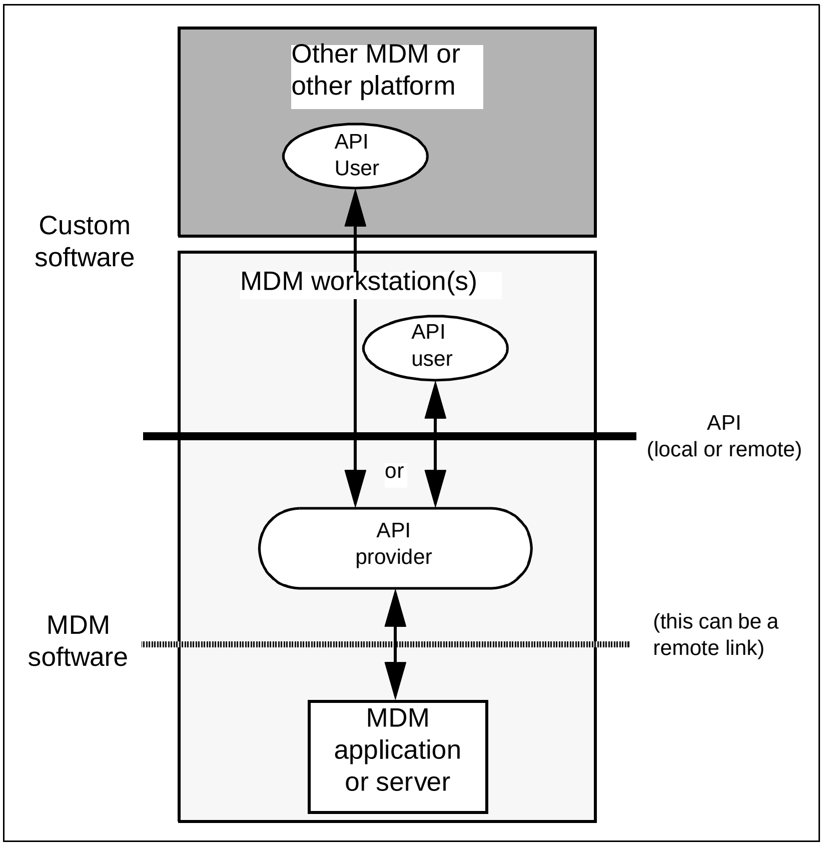
- get data as required
- set certain data, such as provisioning data
- filter the data before receiving it
- be notified when certain events occur

The API environment

As shown in the figure “The API environment” (page 17), customer-developed network management applications can reside

- on the same Preside Multiservice Data Manager (MDM) platform as the accessed MDM application
- on another MDM or equivalent platform
- in a custom application environment representing a completely different technology than the MDM platform

Figure 1
The API environment



The API functional model

The API Provider is the software that provides access to Preside Multiservice Data Manager (MDM) data. The API User is the script, program, or human user that communicates with the API Provider over the interface. The API is defined in terms of the messages passed between the API User and the API Provider.

In a typical session between an API Provider and an API User, the API User issues requests and receives responses and/or errors from those requests. In addition, the API Provider may send event reports to the API User.

An API is implemented using UNIX stdin, stdout, and stderr. API messages (encoded as ASCII characters) are received by the API Provider from stdin (or from a file), and sent to stdout (or to a file). Error messages from the API Provider are sent to stdout. Trace and debugging information is sent to stderr. This allows for many methods of interfacing the API User to the API Provider.

API Providers do not provide connection capabilities. The developers of API User software determine the optimum mechanism for accessing the API Provider. The communication paths used for various configurations are described in detail in "Connecting your custom program to an API" (page 71).

Hardware

The hardware required to use any API is not restricted to a particular type or manufacturer.

Software

The API Provider software required to use any API is supplied by Nortel Networks, and exists on the Preside Multiservice Data Manager (MDM) workstation. The API User software is supplied by the customer, and can exist on the MDM workstation or on another platform.

Migration and compatibility

A migration path will be provided for users of the API to move to future versions of the API with minimal disruption."Extensibility rules" (page 62) provides information to simplify this migration.

Chapter 2 Current APIs

This section explains the different Application Programming Interfaces (APIs) that are available and the data you can manipulate with each. This section contains the following information:

Network Model API

The Network Model API is an ASCII interface that provides access to the following:

- state of each network component
- topology of the network model
- attribute value information for each network component
- possible types of network components
- possible attribute types for each network component
- possible values for each attribute of the possible network component types

- hierarchical structure of the network component types
- notification of network and raw state changes for each network component
- notification of network-wide changes, such as network component creation and deletion

For further information on the Network Model API, see 241-6001-201 *Preside MDM Network Model API Reference Guide*.

Alarm and Status API

The Alarm and Status API is an ASCII interface that provides access to the following:

- recent Preside Multiservice Data Manager (MDM) alarms.
  The alarm information is presented in a format that is common to both DPN and Passport.
- current raw state of the DPN and Passport components
- recent DPN status records
- notification of alarm, status, and raw state change events

Once acquired through the API, the information is available for manipulation and presentation by the API User. For further information on this API,see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*.

DCD API

The data collection daemon (DCD) server has specific API requests; it is similar to the Alarm and Status API. The DCD API can be used to

- discover the database contents of components
- monitor DCD output, including alarms, state change notifications, and server notifications

For more information DCD API, see 241-6001-118 *Preside MDM SNMP Surveillance Adapter Guide*.

DPN Provisioning API

The DPN Provisioning API is an ASCII interface that allows you to create, view, and modify service data found in Preside Multiservice Data Manager (MDM) Configuration Component Provisioning information. The Provisioning API reads service requests from standard input (stdin) and writes responses to standard output (stdout).

For further information on this API, see 241-6001-204 *Preside MDM DPN Provisioning API Reference Guide*.

Passport Provisioning API

The Passport Provisioning API is an ASCII interface that allows you to create, view, and modify service data for Passport switches that is found in MDM Configuration Component Provisioning information. The Provisioning API reads service requests from standard input (stdin) and writes responses to standard output (stdout).

For further information on this API, see 241-6001-207 *Preside MDM Passport Provisioning API Reference Guide*.

Provisioning Command Filter API

The Provisioning Command Filter API is an ASCII interface that provides a means of controlling the provisioning commands that are allowed and those that are not allowed. The blocking of commands is based on a combination of factors including: the command being executed, the specified service data component, and field values.

For further information on this API, see 241-6001-209 *Preside MDM Provisioning Command Filter API Reference Guide*.

Embedded Programming Interface

The Embedded Programming Interface (EPI) enhances the usability of the API interfaces and utilities by providing access to them through the DeskTop Korn Shell (DtKsh) and Tool Command Language (Tcl) scripting languages.

The Preside Multiservice Data Manager (MDM) EPIs provide more powerful and efficient access to the MDM API and Command Access programming interfaces for network operations automation. They allow you to write applications to collect data from and interact with multiple MDM interfaces at the same time (for example, correlating multiple alarm streams, and sending commands to the network elements that are triggered by Network Model state changes or other notifications). These EPIs also make it easier to perform complex API query sequences, where the nature of later queries may depend on the results of previous ones (for example, a recursive descent down parts of the Network Model).

Note: EPIs do not change the API or command syntax.

For further information, see 241-6001-211 *Preside MDM Embedded Programming Interface Reference Guide*.

Chapter 3
Managed object model

This section explains the managed object model, and contains the following information:

About managed objects

An object-oriented model structures the messages passed across an API. Management information is modeled as a set of managed objects. Management interactions across the API are structured as messages to and from the managed objects.

A managed object is an abstraction of a network resource, such as a data switch or a port, or of a network management resource, such as a command log. A managed object has a set of attributes, which are data values describing the state or characteristics of the resource that the managed object represents. It may emit event reports to notify the API User of some event, such as a device failure. A managed object may accept requests to perform specified actions.

A managed object can represent, for example, a Packet Module (PM), Peripheral Interface (PI), link, organization, alarm record, sieve, or connection. Managed objects can represent physical or logical resources, and static or dynamic resources.

Managed objects are organized into classes. All the managed objects of a given class have the same set of attributes (with or without optional attributes), and support the same set of actions, event reports, and behaviour. A specific managed object of a given class is sometimes referred to as a managed object instance.

Attributes

An attribute of a managed object holds a value or a set of values of a given data type. Attributes represent properties of the managed object or information about the managed object, such as its name or other identifiers, its state, its configuration, and related managed objects. Attributes can also carry data values in action requests and responses, in event reports, and in other API messages.

Attributes can represent static or dynamic data, and can represent data that exists in a physical network resource, or data that exists only for management purposes. Some attributes are as follows:

- raw state (of a component)
- Network Administration and Management System identifier (NAMS ID)
- connected links
- annotation (text that describes the purpose of a sieve)
- node type
- shelf type

Attributes of an object class can be mandatory or optional. All instances of an object class must include all the mandatory attributes and may include some or all of the optional attributes.

Each attribute is identified with an attribute name. The same attribute name is used for attributes of the same type that appear in many managed objects of many object classes.

An attribute can hold a single value or a set of values where all values of the set are of the same type. Set-valued attributes have some additional capabilities, such as the ability to add or remove values from the set.

API data types

An attribute has one value or a set of values of a specified type. Attribute values are encoded (using printable ASCII characters) for transmission across the API as a triplet:

```
<attr name> <attr value type> <attr value>
```

APIs use a defined list of data types. The table “API data types” (page 25) lists these data types and the <attr value type> code used in the API, and identifies the way that values of the type are encoded, showing both valid and invalid examples.

An attribute that holds a set of values of one of these types is called a set-valued attribute. There is no special encoding in the API for set-valued attributes; each attribute value is sent separately as a triplet. For further information, see “Long lines” (page 46).

Table 1
API data types

Data Type	Attribute Value Type	Encoding

Boolean	B	ASCII 0 (for FALSE) or ASCII 1 (for TRUE)
		Example
		`B   0`
		`B   1`
		`B   F                    -- invalid`

Integer	I	ASCII decimal representation, optionally preceded by + or - whose length is four bytes or less
		Example
		`I   154`
		`I   -9090`
		`I   one                  -- invalid`

(Sheet 1 of 5)		

Table 1 (continued)
API data types

Data Type	Attribute Value Type	Encoding

Long Integer	L	ASCII decimal representation whose length is greater than four bytes but less than or equal to eight bytes. Unlike the integer (I), the long integer (L) does not include a + or - sign.
		`L 12345677`
		`L 174453`
		`L one                    -- invalid`
		`L +154                   -- invalid`

Hexadecimal	H	ASCII characters 0-9 and A-F
		Example
		`H   A9012FF909E`
		`H   45E1AX89            -- invalid`

Date/Time	D	ASCII characters following the format:
		yyyy mm dd hh mm ss th
		where yyyy is the year, mm is the month (01-12), dd is the day (01-31), hh is the hour (00-23), mm is the minute (00-59), ss is the second (00-59), t is tenths of a second (0-9) and h is hundredths of a second (0-9). The tenths and hundredths are optional, and some API Providers may ignore them on input.
		When an API Provider that supports the th field compares two D values, one with the th specified, and one without, the missing th is assumed to be 00.
		Example
		`D   1998 04 01 09 10 01 10`
		`D   1998 04 01 09 10 01`
		`D   98 04 01 09 10 01 -- invalid`

(Sheet 2 of 5)		

Table 1 (continued)
API data types

Data Type	Attribute Value Type	Encoding

String	S	ASCII characters, excluding carriage returns, tabs, line-end, form feeds, backspaces, and quotes, optionally quoted with " characters.
		
		Example
		`S   Framework`
		`S   "API Framework"`
		`S   "     API Framework"`
		`S   "API Framework     "`
		`S   API Framework`

Enumerated	E	An ASCII string, like the S type, except that spaces are not allowed, quotes are never used, and case is insignificant. Values are defined for the attribute, and are validated.

Formatted String	FS	ASCII characters, optionally quoted (with " characters). Some special encodings are defined, and must be used if these characters appear in the string:
		
		<line-end> is encoded as \n
		\ is encoded as \\\\
		" is encoded as \\"
		<return> is encoded as \r
		<form feed> is encoded as \f
		<tab> is encoded as \t
		<backspace> is encoded as \b
		
		Example
		`FS  API`
		`FS  "API Framework"`
		`FS  "NMS\n\"API\"\nFramework"`
		`FS  "This is a quote: \""`
		`FS  "This is a single backslash: \\"`

(Sheet 3 of 5)		

Table 1 (continued)
API data types

Data Type	Attribute Value Type	Encoding

Sequence of Strings	SS	A sequence of zero or more string values (of the S type) delimited by spaces.
		
		Example
		`SS  API Framework -- two strings`
		`SS  "API" Framework -- two strings`
		`SS  NMS "API Framework" -- two strings`
		`SS  NMS " API " Framework -- three strings`
		`SS  "NMS API Framework" -- one string`

Sequence of Integers	SI	A sequence of zero or more integer values (of the I type) delimited by spaces.
Sequence of Booleans	SB	A sequence of zero or more boolean values (that is, 0s and 1s of the B type) with no separators between them.
Range of Strings	RS	Two string values (of the S type) delimited by spaces. If the two values are a and b, the range is defined as follows: if a ð b, x is in the range if x Š a AND x ð b if a > b, x is in the range if x < a OR x > b String comparisons are normal lexical ASCII ordering, but specific attributes may define the precise ordering rules.
Range of Integers	RI	Two integer values (of the I type) delimited by spaces. If the two values are a and b, the range is defined as follows: if a ð b, x is in the range if x Š a AND x ð b if a > b, x is in the range if x < a OR x > b
Range of Date/Time	RD	Two date/time values (of the D type) delimited by a hyphen (which can be surrounded by spaces). If the two values are a and b, the range is defined as follows: if a ð b, x is in the range if x Š a AND x ð b if a > b, x is in the range if x < a OR x > b
(Sheet 4 of 5)		

Table 1 (continued)
API data types

Data Type	Attribute Value Type	Encoding

Node Identifier	NI	Encoded like an SS value, except that the strings in the sequence are paired (for example, category/key value pairs). Values are never quoted, and spaces are not permitted within each string.
		
		Example
		`NI  PM 01 PE 3 PI 3`
		`NI  PM/01 PE/3 PI/3 -- invalid`
		`NI  "PM" "01" "PE" "3" "PI" "3" -- invalid`
		`NI  PM "01" PE "3" PI "3" -- invalid`

Link Identifier	LI	Encoded using the structure S:NI:NI: The S is a string that indicates the link type, and the NIs are node identifiers that indicate the endpoints of the link. The API Provider accepts the two NIs (endpoints) in either order, but always generates the smallest first. **Example** `LI NL:PM S01 PE 1 PI 1 PO 1:PM S02 PE 1 PI 1 PO 2:`
(Sheet 5 of 5)		

Naming

The set of managed objects visible through an API are arranged in a single hierarchy with a single root, called the containment hierarchy. The name of a managed object is based on its position in this hierarchy, and is represented as an attribute of the managed object, the naming attribute.

Two managed objects joined by an arc of the containment hierarchy are called the superior and the subordinate in the context of their relationship to each other. The superior is the managed object closer to the root. A managed object usually has only one superior, but can have more than one. No managed object is allowed to be the superior of itself, directly or indirectly.

The root is a virtual object that is the ultimate superior of all managed objects in the containment hierarchy. The root has a null name of rootId NI and an object class of root.

The containment hierarchy is arranged according to the containment relationship. In some cases, the containment relationship between a superior and subordinate is very clear and physical (for example, a PE is contained in a PM). In other cases, the containment relationship is based on a more logical relationship or is artificially chosen to simplify access to groups of related managed objects.

The name of a managed object is sent across the API as a triplet (in the same form as an attribute):

```
<attr name> <attr value type> <attr value>
```

This attribute, called the naming attribute, is an attribute of the named managed object, but it cannot be retrieved as an attribute by the API User; it is only used for naming and filtering (if the API supports filtering). No two existing managed objects can have the same name.

Constructing the three parts of the name for a new managed object

1 Determine the <attr name>, based on the object class.

- for sieves, use the sieveId attribute
- for node type objects, use the nodeTypeId attribute
- for nodes, links, and service data components, use the compId attribute

2 Determine the <attr value type> for this attribute.

In most cases, there is only one <attr value type> for a given name attribute, but in some cases, the <attr value type> also depends on the object class.

- for sieveId, use I (integer)
- for nodeTypeId, use S (string)
- for compId, use NI (node identifier) for nodes and service data components, or LI (link identifier) for links

3 Select an <attr value> for the managed object.

The <attr value> can be based on the position of the managed object in the containment hierarchy, relative to some managed object. The method of choosing an <attr value> depends on the object class, and is specified in detail in the documentation for each API.

- for sieves, use an integer that is unique for all existing sieve objects
- for node type objects, use a string that is unique across all existing node type objects
  Example “PM” or “EM-FRAMER”
- for nodes, use a node identifier, consisting of category/value pairs starting at the module
  Example PM R66 PE 1 PI 2
- for links, use a link identifier, consisting of the link type and the two endpoint node identifiers
  Example NL:PM R66 PE 1 PI 2 PO 1:PM R33 PE 2 PI 1 PO 1:
- for service data components, use a node identifier, consisting of category/key value pairs starting at the module (if no upload action has been performed) or at the uploaded component
  Example PE 1 PI 2

Locating the managed object within the containment hierarchy

1 Determine a specific naming context object, based on the object class and <attr name>.

The naming context object is a managed object that is the superior (directly or indirectly) of all managed objects with the given object class and <attr name>.

- for sieve objects, use the root
- for node type objects, use the type set object for node types, which is immediately subordinate to the root
- for a node object, determine the node type: for modules and their subcomponents, use the physical root object; for organizational nodes, use the organizational root object (both of which are immediately subordinate to root)
- for a link object, determine the link type: for physical links, use the physical root object; for organizational links, use the organizational root object
- for a service data component, use the managed object at which the service data was uploaded

2 Find the managed object within the sub-hierarchy under the naming context object, based on the <attr value>.

The <attr value> may traverse the containment hierarchy step by step starting at the naming context object, or it may simply give a value that is unique under the naming context object, or some combination of these two approaches.

Example

There is only one sieve object with a sieveId of 7, and it is immediately below the root.

The node type object with a nodeTypeId of EM-FRAMER is somewhere in the containment hierarchy below the EM node type object, which is immediately below the type set object for node types.

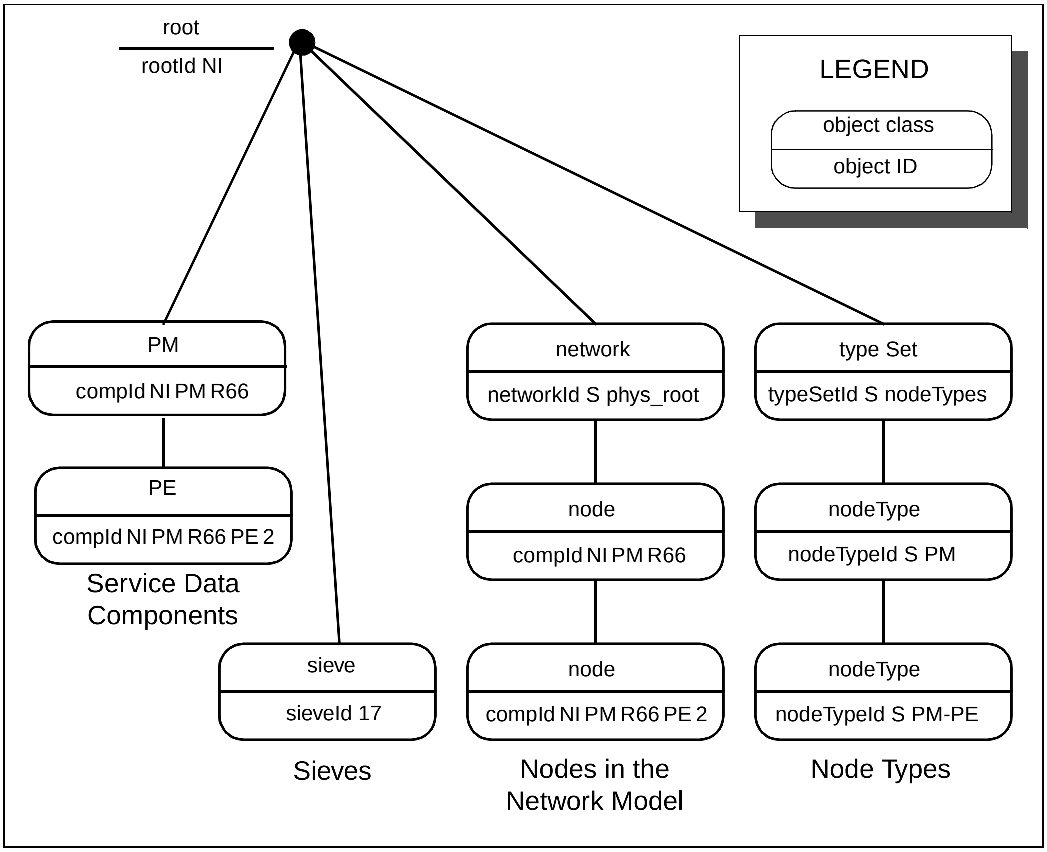
The node object with a compId of PM R66 PE 1 is immediately below the PM R66 node, which is immediately below the physical root object.

Sample containment hierarchy

The figure “Sample containment hierarchy” (page 33) shows a sample containment hierarchy, with the names of some of the managed objects. Note that this diagram does not illustrate a real containment hierarchy for an API; it includes selected managed objects from several APIs to show some of the possible name structures.

The object naming attribute is shown in the containment hierarchy diagram in the lower portion of the object class box. For example, the object ID naming attribute of the PM object class is compId as shown in the figure “Sample containment hierarchy” (page 33).

Figure 2
Sample containment hierarchy



Event reports

Some managed objects send event reports across the API from the API Provider to the API User. An event report can include a set of attributes that describe the event. Some event reports are:

- alarm
  indicates a failure in the network or the correction of a previous failure
- state change
  indicates a changed value for a state attribute of a managed object

- status
  provides periodic statistical data

The definition of an event specifies the name of the event, the conditions that result in the event report being sent, and the attributes that are included in the event report.

The sieve object controls the flow of event reports from the API Provider to the API User. The API User may configure sieve objects to pass on all event reports, no event reports, or only event reports meeting certain conditions. For further information, see “Sieve object class definitions” (page 63).

Actions

The API User can invoke an action on some managed objects. An action is a defined procedure carried out by the managed object. Actions are used when the pre-defined API operations (for reading/writing attributes and creating/ deleting managed objects) are not sufficiently powerful or semantically appropriate for the required functions.

An action has an action type and may be defined with attributes as parameters to the request and/or attributes as parameters to the response.

Selecting multiple managed objects

You can direct the same message to a set of managed objects using scoping and filtering.

Select a set of managed objects to receive a request:

1. Specify the object class and the name (object ID) of a base managed object for the request.
2. If desired, specify a scope.
   See “Scope” (page 35).
3. If desired, specify a filter.
   See “Filters” (page 36).

Each managed object within the specified scope, and for which the filter evaluates to TRUE, receives the request, processes it, and sends a response or an error message across the API. The API Provider sends an end-of-response message.

The end-of-response message is always sent, even if no managed objects are selected by the scope and filter process, and even if one or more error messages were returned. If the API Provider implementation requires that no responses are sent after an error message is sent, the first error message is always followed immediately by an end-of-response message.

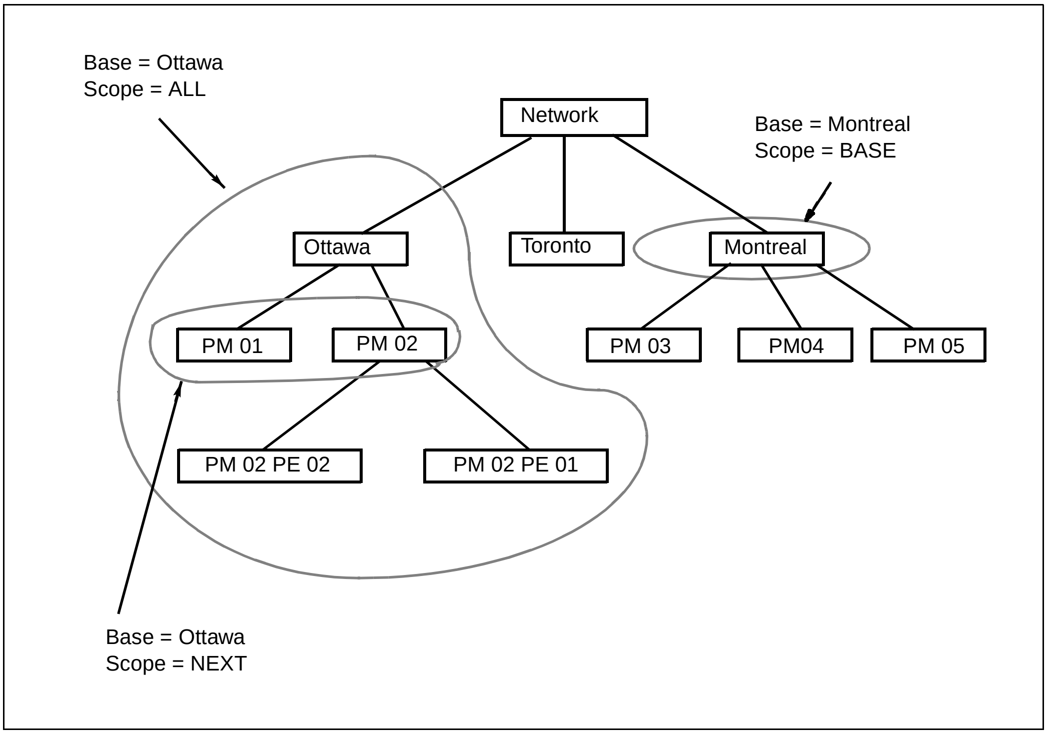
Scope

The scope identifies a subset of the containment hierarchy starting from a base managed object. The default for scope is base. The base managed object is the root of the subset. As shown in the figure “Scope” (page 35), the scope can be one of the following:

- BASE
  Only the base managed object is selected. This is the default scope.
- ALL
  All managed objects in the entire subset from the base managed object down are selected.
- NEXT
  All managed objects immediately subordinate to the base managed object are selected, but the base managed object is not selected.

A scope can select the same managed object more than once because managed objects can have multiple superiors. If a managed object is selected more than once, the API Provider determines whether that managed object performs the operation and responds once or multiple times.

Figure 3
Scope



Filters

A filter is a logical test applied to the attributes of each managed object selected by the scope. If no filter is specified, all managed objects within the scope are selected. In a request message, a filter is specified as a set of filter lines, each of which contains an operator and an attribute name, data type and value. Filter operators are described in the table “Filter operator descriptions” (page 37). The data type in the filter line should match the data type for the attribute. The test in one filter line applies to a single attribute.

Table 2
Filter operator descriptions

Operator	Meaning	Description
P	present	no attribute type and value are specified; TRUE if the attribute is present in the managed object
EQ	equal to	TRUE if the attribute value is equal to the specified value
NE	not equal to	TRUE if the attribute value is not equal to the specified value
LT	less than	TRUE if the attribute value is less than the specified value
GT	greater than	TRUE if the attribute value is greater than the specified value
LE	less than or equal to	TRUE if the attribute value is less than or equal to the specified value
GE	greater than or equal to	TRUE if the attribute value is greater than or equal to the specified value
LEFT	includes left substring	TRUE if the attribute value includes the specified value as a left-aligned substring
MIDDLE	includes middle substring	TRUE if the attribute value includes the specified value anywhere as a substring
RIGHT	includes right substring	TRUE if the attribute value includes the specified value as a right-aligned substring
IN	in range or sequence	TRUE if the attribute value is in the specified range or sequence of values
NOTIN	not in range or sequence	TRUE if the attribute value is not in the specified range or sequence of values

For filter operators EQ, NE, LEFT, MIDDLE, and RIGHT, attributes and values with the same data type can be compared, and an attribute of type NI can be compared with a value of type S. For the comparison, the value of the type NI attribute is converted to a value of type S by inserting a single space between each of the strings in the NI value, and removing any spaces at the beginning or end, but, the attribute itself is not changed.

Example
```
_filter: compId LEFT S PM R
```

```
Where:
compId = <attr_id>
LEFT = <operator>
S = <data_type>
PM R = <attr_value>
```

matches all managed objects whose compId attribute starts with PM R, for example, PM R66, or PM R21 PE 1.

For IN and NOTIN, a string attribute can be compared against a value of type RS or SS, and an integer attribute can be compared against a value of type RI or SI, and a date/time attribute can be compared against a value of type RD.

If a filter line refers to an attribute that is defined to be set-valued, the specified filter test is applied separately to each value of the attribute, and if it passes for one or more values, the filter line evaluates to TRUE.

When multiple filter lines are supplied in the request, they are combined as follows:

- Filter lines that refer to the same attribute are logically ORed together.
- These results are then logically ANDed together.

Example
```
_filter: problemState EQ S DOWN
_filter: problemState EQ S TROUBLED
_filter: lineSpeed GT I 19200
```

selects all managed objects within the scope that have both a problemState and a lineSpeed attribute, where problemState is either DOWN or TROUBLED, and lineSpeed is greater than 19200.

A filter line evaluates to FALSE if the specified attribute is not present in the managed object. Together with the AND and OR rules, this means that a managed object is selected only if all of the attributes specified in the filter lines are present in the managed object.

In addition to the normal attributes of a managed object, a filter line can apply to the object's class or naming attribute, even though these two attributes are not retrievable. Some APIs may not support this capability.

Chapter 4 API messages

This section describes API messages, and contains the following information:

About API messages

API Providers and API Users communicate with each other by exchanging messages. A message is a group of one or more non-blank lines followed by a single blank line. Each message line provides information that identifies the service and the target data object.

API sessions consist mainly of exchanging requests and responses.

Message types

The message types are as follows:

- request
  The API User sends requests. The generally supported requests are GET, SET, ACTION, CREATE, DELETE, REGISTER, and DEREGISTER.
- response
  The API Provider replies to requests with zero or more responses.

- error
  The API Provider replies to requests with zero or more error messages.
- block construct
  The API Provider embeds replies in a block construct, which forms a part of the surrounding message, when a response or an error message includes an attribute that has a multi-line value.
- end-of-response
  The API Provider completes its response or error message reply with an end-of-response message.
- event report
  The API Provider sends event report messages to report the occurrence of events.
- version
  When an API session is established, the API Provider sends a message indicating the version of the API.
- end
  Either the API User or the API Provider can send an end message to terminate the API session.
- trace
  The API User can toggle trace message for debugging.
- echo
  The API User can toggle echo message for debugging.

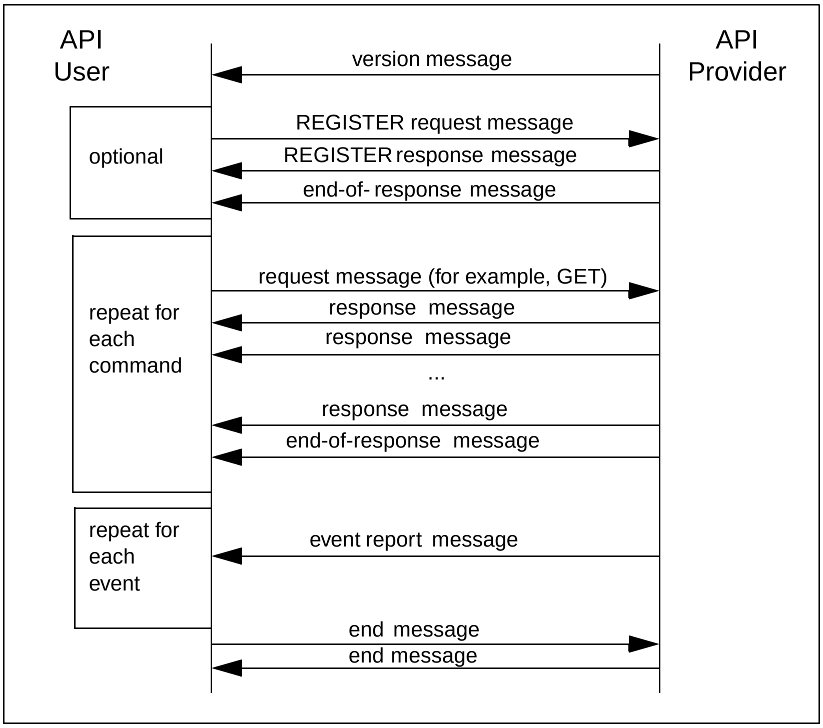
Message sequence

Some API Providers operate synchronously, others operate asynchronously. In synchronous operation, the API Provider processes each request fully and returns all responses and/or error messages and the end-of-response message before processing the next request. In asynchronous operation, the API Provider processes multiple requests concurrently, and multiple responses from different requests are interspersed.

Event reports can be sent by the API Provider at any time, including in between the responses to a request. However, each message from the API Provider (response, error, end-of-response, or event report) is sent in its entirety; a message is never sent in the middle of another message.

The figure “API session time-line” (page 43) shows the order of messages in a typical API session.

Figure 4
API session time-line



General message structure

An API message is a sequence of message lines, and is terminated by a blank line. All message lines are encoded as ASCII printable characters. The lines in an API message must appear in the order that they are specified in “Message structures” (page 48).

Message lines

A message line consists of a line label, a colon, one or more spaces and/or tabs, the parameters of the line, separated by spaces and/or tabs, and a line-end character, in that order.

A line label is a string that begins with an underscore. The accepted line labels for input and output are shown in the table “Message line labels” (page 44). No abbreviations are supported.

Table 3
Message line labels

Line label	Line purpose/contents	Into/Out of API Provider
`_action_type`	action type	I
`_attr`	attribute value	I/O
`_attr_id`	attribute name	I
`_block`	block construct	O
`_capability`	capability set	O
`_cmd`	command name	I
`_echo`	echo message	I
`_end`	end message	I/O
`_end_block`	end of block construct	O
`_end_resp`	end-of-response message	O
`_error`	error message	O
`_event_type`	event type	O
`_filter`	filter	I
`_inv_id`	invoke identifier	I/O
`_mod`	modification	I
`_obj_id`	object name	I/O
`_obj_class`	object class	I/O
`_password`	password	I
`_ref_obj_id`	reference object name	I
`_scope`	scope	I
`_sieve_id`	sieve identifier	O
(Sheet 1 of 2)		

Table 3 (continued)
Message line labels

Line label	Line purpose/contents	Into/Out of API Provider
`_sup_obj_id`	superior object name	I
`_time`	time	O
`_trace`	trace message	I
`_user_id`	user ID	I/O
`_version`	version message	O
(Sheet 2 of 2)		

Comments

An API message line that starts with an asterisk (*) or an octothorpe (#) is treated as a comment.

Upper and lower case

Generally, the API Provider accepts either upper or lower case on input. More specific information is provided in the table “ASCII character case for input and output” (page 45).

Table 4
ASCII character case for input and output

Item	Input case	Output case
line labels	insignificant	lower with underscores
keywords	insignificant	upper with underscores
object classes	insignificant	mixed with no underscores or hyphens
attribute names	insignificant	mixed with no underscores or hyphens
attribute value types	insignificant	upper
(Sheet 1 of 2)		

Table 4 (continued)
ASCII character case for input and output

Item	Input case	Output case
attribute value type E	insignificant	significant
attribute value type H	insignificant	upper
attribute value type S, FS, SS, RS, NI, LI	significant	same case as input
action types	insignificant	mixed with no underscores or hyphens
event types	n/a	mixed with no underscores or hyphens
<user id> parameter	significant	n/a
<password> parameter	significant	n/a
command line echo	n/a	same case as input
(Sheet 2 of 2)		

Quotes

On input, the API Provider accepts values of type S, SS, FS, or RS with or without quotes. The presence or absence of quotes has no effect on upper and lower case handling.

On output, strings may or may not contain quotes.

Values of type NI, LI, H, and E never contain quotes. This means that categories, key values, and link types cannot contain spaces.

Long lines

Some environments have line length limitations. If an API message exceeds that length, it must be split and carried across the interface as separate lines.

To split a long line, insert a backslash character (\) followed by a line-end character. The backslash indicates that the following line-end is not a significant character in the API message. This method can be applied to messages moving in either direction.

Example 1

```
S   "This is a single-line\
    string value encoded using two lines."

S   "This string has an embedded backslash \\
    which happens to fall at the end of a line\
    but not at the end of the data value."
```

If the last character of a data value is a backslash, the last string of the data value must be in quotes.

Example 2

```
FS "E-S101: PE 7 PI 7 PO 5 ITI 0 ITILINK 0: \"IN SPEED\
   \" not equal to \"Out \n Speed\" when speed is 110\
   or AUTO"

S  "backslash\"                        -- valid
S  back\slash                          -- valid
SS This is a backslash: "\"            -- valid
NI PM 01 PE 2 xy "3\"                  -- valid

S  backslash\                          -- invalid
SS This is a backslash: \              -- invalid
NI PM 01 PE 2 xy 3\                    -- invalid
```

On output, an attribute value that takes up multiple lines can be sent as multiple lines if it is enclosed in a block construct. A block construct takes the place of a single line in an API message. The format of a block construct is given in “Block construct” (page 61).

Invoke identifiers

If an API Provider can handle requests asynchronously, responses can be returned in a different order than the requests. Also, a request can result in multiple responses, which could be interspersed with the responses of other requests. The invoke identifier allows an API User to correlate received responses with the issued requests. If you place an invoke identifier in a request, all responses, error messages, and the end-of-response message for that request carry the same invoke identifier.

The invoke identifier is an integer. You must ensure that the invoke identifier of a request in progress is not reused in a new request.

Message structures

The following subsections describe the structure of the individual API messages. “Message syntax summary” (page 79) provides a summary of the syntax of the API messages in a quick-reference format.

> ***Note:*** Review the containment hierarchy diagram for object class and object id when writing APIs.

GET

The GET command retrieves values of attributes from managed objects. A GET request results in zero or more GET responses (one for each selected object), followed by an end-of-response message.

GET request

```
_cmd: GET
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
[_scope: BASE | NEXT | ALL]
[_filter: <attr name> <operator>
[<attr value type> <attr value>]]
…
[_attr_id: <attr name> | ALL]
…
<blank line>
```

The _obj_class line gives the object class of the base object. The _obj_id line gives the name of the base object. The _scope and _filter lines specify the selected managed objects for the request, as described in “Selecting multiple managed objects” (page 34).

Each _attr_id line identifies an attribute that you want to retrieve. For each selected object, the requested attributes that are present in the object are returned. It is not an error for some or all of these attributes to be absent from a selected object—each object returns the available attributes (possibly zero).

If a line reading _attr_id: ALL appears in the request, all attributes of the selected objects are returned. If there is no _attr_id line in the request, each selected object responds, but with no _attr lines.

GET response

```
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

One _attr line is present for each returned attribute value. The values of a set-valued attribute are returned in a contiguous set of lines, all with the same <attr name>, and with one value for each line.

SET

The SET command modifies the values of attributes of managed objects. A SET request results in zero or more SET responses (one for each selected object), followed by an end-of-response message.

The API Provider works in one of two ways. When multiple managed objects are selected, the API Provider performs the SET command either atomically or non-atomically as follows:

- atomically
  All of the attributes are modified and no attribute values are returned, or none of the attributes are modified and an error message is returned.
- non-atomically
  Only those attributes that could be modified are modified. The modified attribute values are returned in the SET responses.

SET request

```
_cmd: SET
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
[_scope: BASE | NEXT | ALL]
[_filter: <attr name> <operator>
[<attr value type> <attr value>]]
…
[_mod: <modify operator> <attr name>
[<attr value type> <attr value>]]
…
<blank line>
```

The _obj_class line gives the object class of the base object. The _obj_id line gives the name of the base object. The _scope and _filter lines specify the selected managed objects for the request, as described in “Selecting multiple managed objects” (page 34).

Each _mod line specifies a change to be made to an attribute. An error message is returned if an attribute mentioned in a _mod line is not present in a selected object. The <modify operator> indicates the type of change:

- ADD
  Add a value to a set-valued attribute.
- DEL
  Delete a value from a set-valued attribute.
- DEF
  Set an attribute to its default value or set of values, (in the case of a set-valued attribute). No <attr value type> and <attr value> are required. Set-valued attributes have a default value of the empty set wherever possible to simplify setting the attribute to the null set.
- REP
  Replace the attribute value or set of values with the specified value.

There is no explicit means for replacing a set-valued attribute with a new set of values. Other operations must be combined.

Example
Use a DEF followed by a number of ADDs to give a new set of values to an attribute that has a default value of the null set.

SET response

```
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

If attribute values are to be returned, one _attr line is present for each value of each modified attribute. All of the values of a modified set-valued attribute are returned, not just those that were present in the request. The values of a set-valued attribute are returned in a contiguous set of lines, all with the same <attr name>, and with one value on each line.

ACTION

The ACTION command asks one or more managed objects to perform a defined action. An ACTION request results in zero or more ACTION responses (one for each selected object), followed by an end-of-response message.

ACTION request

```
_cmd: ACTION
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
[_scope: BASE | NEXT | ALL]
[_filter: <attr name> <operator>
[<attr value type> <attr value>]]
…
_action_type: <action type>
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

The _obj_class line gives the object class of the base object. The _obj_id line gives the name of the base object. The _scope and _filter lines specify the selected managed objects for the request, as described in “Selecting multiple managed objects” (page 34).

The _action_type line indicates the type of action that is requested.

The _attr lines supply parameters to the action, if there are any. If a set-valued attribute is used as an action parameter, all of its values appear on contiguous lines.

ACTION response

```
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
```

```
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

The _attr lines give result parameters of the action. If a set-valued attribute is used as an action result parameter, all of its values appear on contiguous lines.

CREATE

The CREATE command creates a single, new managed object. A CREATE request results in a single CREATE response, followed by an end-of-response message.

CREATE request

```
_cmd: CREATE
[_inv_id: <invoke-id>]
_obj_class: <object class>
[_obj_id: <attr name> <attr value type> <attr value>]
[_sup_obj_id: <attr name> <attr value type>
<attr value>]
[_ref_obj_id: <attr name> <attr value type>
<attr value>]
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

The _obj_class gives the object class of the managed object to be created. The _obj_id line specifies the name of the new managed object. The _sup_obj_id line specifies the name of the superior of the new managed object. If the _obj_id is not supplied, the API Provider assigns a name for the new object (or returns an error if it cannot do so).

The _ref_obj_id line specifies the name of an existing managed object from which attribute values are to be copied for the new object. The referenced object must be of the same object class, or at least have a subset of the attributes of the new object.

You can specify initial values for some or all of the attributes of the new managed object, using _attr lines. The values of a set-valued attribute are provided in a contiguous set of lines, all with the same <attr name>, and with one value on each line.

CREATE response

```
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
<blank line>
```

The CREATE response includes the name of the newly-created managed object in the _obj_id line.

DELETE

The DELETE command deletes one or more managed objects. A DELETE request results in zero or more DELETE responses (one for each selected object), followed by an end-of-response message.

DELETE request

```
_cmd: DELETE
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
[_scope: BASE | NEXT | ALL]
[_filter: <attr name> <operator>
[<attr value type> <attr value>]]
…
<blank line>
```

The _obj_class line gives the object class of the base object. The _obj_id line gives the name of the base object. The _scope and _filter lines specify the selected managed objects for the request, as described in “Selecting multiple managed objects” (page 34).

All selected objects are deleted, and every object that is subordinate to the selected object is also deleted, unless the subordinate has another superior that is not being deleted. If the relevant NAME BINDING template has the ONLY-IF-NO-CONTAINED-OBJECTS keyword, an object is only deleted if it has no subordinates.

DELETE response

```
[_inv_id: <invoke-id>]
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
<blank line>
```

The DELETE response includes the name of the deleted managed object in the _obj_id line.

REGISTER and DEREGISTER

The REGISTER command identifies an API User and determines its privileges for the use of the API. A REGISTER request results in a single, optional REGISTER response, followed by an end-of-response message.

The DEREGISTER command terminates the registration of an API User and reverts to default privileges for the use of the API. A DEREGISTER request results in a single, optional DEREGISTER response, followed by an end-of-response message.

If an API supports either of these messages, it must support both.

When an API session is established, an API-specific default set of privileges is assigned to the API User. A REGISTER command changes the set of user privileges. You must issue a DEREGISTER command before issuing another REGISTER command. The DEREGISTER command is not required before terminating an API session (terminating the session automatically deregisters the API User).

Registering and deregistering have no effect on the presence of managed objects in the API Provider's object model, but they may affect the API User's ability to access objects.

Example

The API User receives event reports as a result of using a sieve. A deregistered API User is not entitled to receive those reports, and the sieve becomes locked and ceases sending event reports. When the API User re-registers and re-establishes the privilege to receive the event reports, the sieve unlocks, and event reports can pass across the API again.

REGISTER request

```
_cmd: REGISTER
[_inv_id: <invoke-id>]
[_user_id: <user id>]
[_password: <password>]
```

```
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

The _user_id and _password lines give a user ID and password for registering the API User.

Additional (API-specific) parameters to the REGISTER request may be sent as attributes.

REGISTER response

```
[_inv_id: <invoke-id>]
[_user_id: <user id>]
[_capability: <capability set>]
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

The user ID and/or capability set assigned to the API User may be returned in the response. These lines indicate the user ID and privilege set that the API User has after registering. The format of the capability set is API-specific.

Additional (API-specific) results of the REGISTER request may be sent as attributes.

DEREGISTER request

```
_cmd: DEREGISTER
[_inv_id: <invoke-id>]
<blank line>
```

Since deregistering always returns the user to the default capability set, the request does not require the values for the userid and/or capability that is currently assigned. To change the capability set, you must deregister back to the default capability and then register for the new capability using the REGISTER request.

DEREGISTER response

```
[_inv_id: <invoke-id>]
<blank line>
```

Since the only message line in the response is optional, the response itself is optional. The API Provider may send nothing or a single blank line as a response.

Event report message

The event report message reports the occurrence of an event in a managed object. An event report message sent by the API Provider, is not responded to, and is not followed by an end-of-response message.

```
_event_type: <event type>
_sieve_id: <attr value>
_obj_class: <object class>
_obj_id: <attr name> <attr value type> <attr value>
_time: <time>
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

The _event_type line indicates the type of event that occurred.

The _sieve_id line gives the integer value of the sieveId attribute of the sieve that is forwarding this event report.

The _obj_class line gives the object class of the object that generated the event report. The _obj_id line gives the name of that object.

The _time line indicates the time that the event occurred, or if this time is not available, the local time at the API Provider. The event report definition (that is, the NOTIFICATION behavior) indicates whether the time zone of this time is GMT, local (to the resource or to the API Provider), or unknown.

Parameters of the event report can be carried in _attr lines. If a set-valued attribute is used as an event report parameter, all of its values appear on contiguous lines.

End-of-response message

The end-of-response message indicates that no more responses are forthcoming from the API Provider, and that the processing of the request is complete.

```
[_inv_id: <invoke-id>]
_end_resp: [<command name>]
_time: <time>
<blank line>
```

The <command name> is the type of message taken from the _cmd line of the request that is being closed. The <command name> must be included unless it cannot be determined from the request, for example, if the request is badly formatted.

The _time line gives the current time, local to the API Provider.

Error message

In place of any response message, the API Provider may return an error message to indicate some failure in the processing of a request. After sending an error message, the API Provider may stop processing the request and send an end-of-response message immediately, or it may continue returning responses, and then an end-of-response.

```
[_inv_id: <invoke-id>]
_error: <error name> [<error title>]
[_attr: <attr name> <attr value type> <attr value>]
…
<blank line>
```

The <error name> is a keyword for the type of error. The list in the table “Error names” (page 58) shows the error keywords, and shows the error names that apply to the command types. The <error title> gives a title of the error, suitable for displaying, but is not used to further qualify the error. Do not depend on the presence or contents of the <error title> to control process flow.

The _attr lines give more information about the error. The following attributes are defined for use in the error message:

- errorSeverity
  An enumerated string that indicates the severity of the error; either Information, Warning, Error, or Fatal_Error
- errorCode
  A string that further identifies the error
- errorApplicationId
  A string that identifies the application where the error occurred or was detected
- errorFacility
  A string indicating a facility related to the error
- errorText
  Additional text to describe the error
- errorInformation
  Other information about the error in unspecified format

Table 5
Error names

Error name	Description	Command support
ACCESS_DENIED	The request exceeds the assigned privileges for the API User.	Supports GET, SET, ACTION, CREATE, DELETE.
APPLICATION_ERROR	Unspecified error. The _attr lines further qualify the error.	Supports GET, SET, ACTION, CREATE, DELETE, REGISTER, DEREGISTER.
UNRECOVERABLE_ERROR	An error from which recovery is not possible, for example: out of memory errors, fork failures, internal inconsistencies, failure to connect to a server, failure to open a necessary file.	
(Sheet 1 of 3)		

Table 5 (continued)
Error names

Error name	Description	Command support
DUPLICATE_OBJECT	The object to be created already exists.	Supports CREATE.
INIT_ERROR	An error occurred during API Provider initialization. The API version number is the first token in <error title>.	Not applicable.
INVALID_ACTION_TYPE	The action type is not recognized.	Supports ACTION.
INVALID_ATTRIBUTE_NAME	An attribute name in the request is invalid.	Supports SET, ACTION, CREATE, REGISTER.
INVALID_ATTRIBUTE_VALUE	An attribute value in the request is invalid.	Supports SET, ACTION, CREATE, REGISTER.
INVALID_FILTER	A filter is invalid or inappropriate for the attribute that is being tested.	Supports GET, SET, ACTION, DELETE.
INVALID_OBJECT_ID	The object name in the _obj_id line is invalid.	Supports GET, SET, ACTION, CREATE, DELETE.
INVALID_OBJECT_CLASS	The object class in the _obj_class line is invalid.	Supports GET, SET ACTION, CREATE, DELETE.
INVALID_SCOPE	The scope is invalid.	Supports GET, SET, ACTION, DELETE.
LOGON_FAILS	The user ID or password is invalid.	Supports REGISTER.
MISSING_ATTRIBUTE_VALUE	No default value is available for an attribute, and the CREATE request did not supply one.	Supports ACTION, CREATE, REGISTER.
MODIFICATION_ERROR	A SET request attempted an illegal modification (for example, an ADD to a single-valued attribute).	Supports SET.
(Sheet 2 of 3)		

Table 5 (continued)
Error names

Error name	Description	Command support
NO_SUCH_OBJECT_ID	The object name in the _obj_id line is valid, but no such object exists.	Supports GET, SET, ACTION, CREATE, DELETE.
OUT_OF_SEQUENCE	A message was sent out of sequence (for example, a REGISTER was required first).	Supports GET, SET, ACTION, CREATE, DELETE, REGISTER, DEREGISTER.
SYNTAX_ERROR	The message was improperly formatted (for example, any error in parsing a received message, if the error can be detected without any knowledge of the object model).	Not applicable.
NOT_SUPPORTED	A message line or keyword was recognized/parsed, but is not supported.	Not applicable.
(Sheet 3 of 3)		

Version messages

The version message is sent once, by the API Provider, when the API session is established. It is not responded to, and it is not followed by an end-of-response message.

```
_version: <version number> <version string>
```

The <version number> is of the form x.y where x and y are the major and minor version numbers of the API Provider software. The version string gives the name of the API and possibly some other information.

End messages

The end message is sent by either the API Provider or the API User to terminate the API session.

The end message sent by the API User has the following format:

```
_end:
<blank line>
```

The end message sent by the API Provider has one of the following forms:

```
_end: USER_REQUEST
<blank line>

_end: FATAL_ERROR
<blank line>

_end: SYSTEM_SHUTDOWN
<blank line>

_end: LOST_CONNECTION
<blank line>

_end: LOST_SESSION
<blank line>
```

Trace messages

A trace message can be sent by the API User to turn tracing on or off. The kind of trace information that is collected and the method of collecting it is API-specific. The API Provider does not respond to this message.

```
_trace: ON | OFF
[<blank line>]
```

Echo messages

An echo message can be sent by the API User to turn echoing on or off. When echoing is on, API messages from the API User are echoed back to the API User as comments. The API Provider does not respond to this message.

```
_echo: ON | OFF
[<blank line>]
```

Block construct

A block construct takes the place of a single line in an API message. It can be used by the API Provider when it sends a multi-line attribute value. Block constructs cannot be sent by the API User.

```
_block: <line label> [<parameter value>] …
<data value>
…
_end_block:
```

The <line label> indicates the line that is being formatted as a block construct. There is no colon following the <line label>. The <parameter value>s follow the syntax and semantics of the line being sent. The last parameter of the line (normally an <attr value>) is carried on one or more lines, and is shown above as <data value>. The _end_block line terminates the block construct.

Since a block construct is embedded in a message, it is not terminated with a blank line. It can only carry ASCII values; the H type must be used for non-ASCII data. Blank lines within a block construct do not terminate the block construct or the message that it is contained in. No lines within a block construct may be wider than 72 characters, since the backslash processing described in “Long lines” (page 46) is not done within a block construct.

Extensibility rules

From one version of an API to the next, controlled changes may be made to the message structure and/or object model. To enable compatibility between versions of APIs in the presence of these extensions, the API Provider and the API User must follow certain rules.

The API Provider will ensure the following:

- All request messages that were valid in the previous version remain valid, and are treated the same way.
- Any extensions to request messages must be optional.
- All response messages and events must include at least what was included before, and this “common” information must have the same meaning.

The API User must ensure that any unrecognized extensions to responses and events (for example, extra lines) are ignored.

For both the API User and the API Provider, if an unrecognized message line is to be ignored, it must be checked for the backslash continuation character to ensure that all associated continuation lines are also ignored.

Chapter 5
Sieve object class definitions

This section describes sieve objects, and contains the following information:

About sieve objects

Sieve objects give the API User control over the flow of event reports from the API Provider. Sieves filter event reports, forwarding some event reports across the API, and preventing others from being sent.

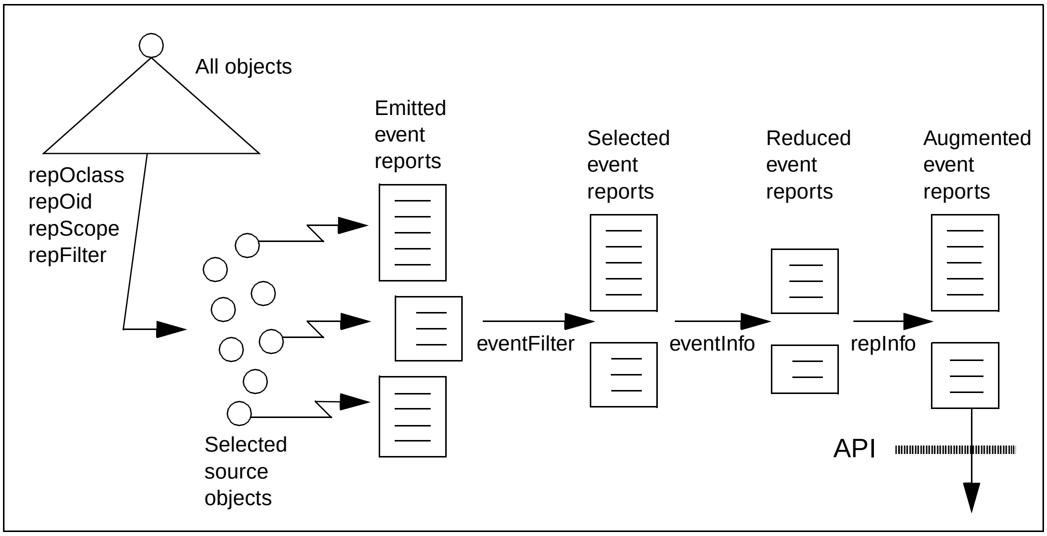
When an API session is established, no sieves are present, so the API User does not receive any event reports until one or more sieve objects are created to forward them. All sieve objects are automatically deleted when the API session terminates.

The API User can create and configure sieve objects to meet its event reporting requirements. A single sieve can be created to forward all events, or separate sieves can be created for event reports of different types and/or severities. Sieve objects are only visible to the API User that created them, and they only forward event reports to that user.

Managed objects send event reports when defined events occur. All event reports are passed to all sieve objects. Each sieve object evaluates a test (the event filter) against the attributes of the event report, and if the event report passes the test, the report is forwarded across the API. The event filter can be applied to the defined attributes of the event report, and optionally to the objectClass, eventType, and naming attributes, using the filter operators defined for each attribute (if any).

The figure “Using sieves for event reporting” (page 64) shows the process of selecting source objects, selecting event reports for forwarding, selecting the attributes of those event reports to include, and adding extra attributes to event reports. Not all APIs support all of these controls, and a given sieve may not include them all.

Figure 5
Using sieves for event reporting



Real time reporting of the following events is supported:

- network state change (reports changes in the values of the stateSeverity and networkState attributes)
- raw state change (reports changes in the value of the rawState attribute)

Real time state notification is supported through the use of the eventReport service. Node and Link class objects emit notifications that are filtered by objects of class Sieve.

The table “Sieve object attributes” (page 65) provides information about the attributes of the sieve object. Only three attributes, sieveId, eventFilter, and admState are mandatory. The other attributes may or may not be used in a particular API. See the appropriate API Reference Guide for details.

Table 6
Sieve object attributes

Attributes	Type	Filter	Description
sieveId	I	EQ, NE, LT, GT, LE, GE	names sieve objects All sieves visible to an API User in a single API Provider have unique sieveIds.
eventFilter	SetOfSS		structured like a selection filter, but the test is evaluated against the attributes of an event report rather than against the attributes of a managed object. See “Filters” (page 36).
admState	I	EQ, NE	determines whether the sieve is locked or unlocked. A value of 0 locks the sieve, 1 unlocks the sieve. A locked sieve does not forward event reports.
annotation	S	EQ, NE, RIGHT, MIDDLE, LEFT	describes the purpose of the sieve **Example** All critical alarms
eventInfo	SetOfS		specifies a list of attributes of the event report that are included in the event report message By default, all event report attributes are included.
repOclass	S		indicates the object class of the base object for use with the repScope attribute The value is as in an _obj_class line. Only event reports from objects selected by this attribute are forwarded. If this attribute is not present, event reports from all objects are forwarded.
repOid	SS		indicates the name of the base object for use with the repScope attribute The value is as in an _obj_id line. Only event reports from objects selected by this attribute are forwarded. If this attribute is not present, event reports from all objects are forwarded.
(Sheet 1 of 2)			

Table 6 (continued)
Sieve object attributes

Attributes	Type	Filter	Description
repScope	E		indicates the scope of event report source objects The value is as in a _scope line. Only event reports from objects selected by this attribute are forwarded. If this attribute is not present, event reports from all objects are forwarded.
repFilter	SetOfSS		gives a filter to apply to event report source objects The value is as in a _filter line. Only event reports from objects selected by this attribute are forwarded. If this attribute is not present, event reports from all objects are forwarded.
repInfo	SetOfS		specifies a set of additional object attributes that are added to event reports forwarded by this sieve. The default causes no attributes to be added.
(Sheet 2 of 2)			

The API User can use the GET, SET, CREATE, and DELETE commands to control the reporting of events by manipulating sieve objects and their contents. The table “Commands permitted on sieve attributes” (page 66) indicates the API commands that are permitted for each sieve attribute.

Table 7
Commands permitted on sieve attributes

Attribute	API commands
eventFilter	GET, CREATE
annotation	GET, CREATE
sieveId	GET, DELETE
admState	GET, SET, CREATE
repInfo	GET, CREATE
eventInfo	
repOclass	GET, CREATE
(Sheet 1 of 2)	

Table 7 (continued)
Commands permitted on sieve attributes

Attribute	API commands
repOid	GET, CREATE
repScope	GET, CREATE
repFilter	GET, CREATE
(Sheet 2 of 2)	

Using sieves

Sieves notify the API User when certain events occur. For example, the API User can create a sieve to note when any new state change events are generated.

Creating a sieve

The Create command creates a sieve that reports changes in the network or raw state.

Example 1

Request all the network and raw state change notifications as follows:

```
_cmd: CREATE
_obj_class: sieve
_attr: eventFilter SS eventType EQ S
networkStateChange
```

When the sieve creation is successful, the response message is as follows:

```
_obj_class: SIEVE
_obj_id: sieveId I 4

_end_resp: CREATE
_time: 1994 02 05 00 19 32
```

Note: In this example, a sieve with a sieveId of 4 has been created.

The format of the networkState change notification report is as follows:

```
_sieve_id: 4
_obj_class: node
_obj_id: compId NI PM A4901 PE 8 PI 8
_event_type: networkStateChange
```

```
_time: 1994 02 05 00 24 37
_attr: networkState E ISTB
_attr: stateSeverity I 1
```

Example 2

Request all the network and raw state change notifications for all the modules under SITE X, as follows:

```
_cmd: CREATE
_obj_class: sieve
_attr: eventFilter SS eventType EQ S
networkStateChange
_attr: eventFilter SS eventType EQ S rawStateChange
_attr: repOclass S orgNode
_attr: repOid SS orgNodeId NI SITE X
_attr: repScope E NEXT
_attr: repFilter SS objectClass NE S link
```

When the sieve creation is successful, the response message is as follows:

```
_obj_class: SIEVE
_obj_id: sieveId I 5

_end_resp: CREATE
_time: 1994 02 05 00 27 16
```

Note: In this example, a sieve with a sieveId of 5 has been created.

The format of the networkState and rawState change notification reports is as follows:

```
_sieve_id: 5
_obj_class: node
_obj_id: compId NI PM A4901 PE 8 PI 8 PO 1
_event_type: rawStateChange
_time: 1994 02 05 00 29 43
_attr: rawState E OOS

_sieve_id: 5
_obj_class: node
_obj_id: compId NI PM A4901 PE 8 PI 8
_event_type: networkStateChange
_time: 1994 02 05 00 30 32
_attr: networkState E ISTB
_attr: stateSeverity I 1
```

Getting the attributes of a sieve

Example

Find all the attributes of the sieve with a sieveId of 6 as follows:

```
_cmd: GET
_obj_class: sieve
_obj_id: sieveId I 6
_attr: ALL
```

The response message is as follows.

```
_obj_class: sieve
_obj_id: sieveId I 6
_attr: annotation S “This is an annotation attribute”
_attr: admState I 1
_attr: eventFilter SS eventType EQ S
NetworkStateChange
_attr: repType S NODE
_attr: repBase SS NAME NI PM R66
_attr: repScope S ALL

_end_resp: GET
_time: 1994 02 05 00 38 09
```

Note: In this example, the admState attribute is 1, meaning that the sieve is unlocked, and it reports events as soon as they occur.

Locking and unlocking a sieve

The only attribute that can be changed is the admState attribute, which locks and unlocks the sieve. When a sieve is locked, the event reporting is stopped.

Use the SET command to change the integer of the admState attribute to 0 (lock the sieve), or 1 (unlock the sieve).

Example

Lock the sieve with a sieveId of 7 (and stop event reporting) by changing the admState attribute to 0 as follows:

```
_cmd: SET
_obj_class: sieve
_obj_id: sieveId I 7
_mod: REP admState I 0
```

The response message is as follows:

```
_obj_class: sieve
_obj_id: sieveId I 7
_attr: admState I 0

_end_resp: SET
_time: 1994 02 05 00 43 12
```

To restart the sieve, set the admState to 1.

Removing a sieve

Example

Delete the sieve with a sieveId of 4 as follows:

```
_cmd: DELETE
_obj_class: sieve
_obj_id: sieveId I 4
```

The response message is as follows:

```
_obj_class: sieve
_obj_id: sieveId I 4

_end_resp: DELETE
_time: 1994 02 05 00 46 29
```

Chapter 6
Connecting your custom program to an API

The API Provider and the API User communicate by exchanging ASCII messages through UNIX pipes using stdin, stdout, and stderr.

The necessary communication connections or piping between the API Provider and the API User must be established at the time of the invocation of the API Provider. This section briefly discusses ways to establish piping. This section contains the following information:

- “Using an API interactively” (page 71)
- “API configured with INETD” (page 75)
- “API and custom program on different platforms” (page 76)
- “API and custom program on the same workstation” (page 76)
- “API and custom program on different workstations” (page 76)

Using an API interactively

You can invoke the API Provider from the terminal without any redirection to use it interactively, since stdin and stdout normally go to the terminal or window. The syntax of the API is not really suited to this sort of use, but it can be a useful mode for learning about the API, testing potential commands, and seeing what responses look like before writing an application program.

If you are using a windowing system to create your custom program, you can edit a file of API message lines, and leave the edit session and its window up while trying out the API interactively in another window. Cut and paste the message lines from the edit session window to the API session window. Build up your API message line file by adding more message lines to the file and making corrections to the message lines that fail.

Controlling an interactive session

In the API session window, you can type and edit only one line of input at a time. When the carriage return is entered, the input line is passed to the API Provider for parsing. After a line is input, the API Provider enters a reading loop that stops all further processing of event notifications. If you were receiving events from a previously created sieve, these events are no longer displayed until your typing is completed.

> ***Note:*** In some APIs, a lone carriage return puts you in a reading loop. If you have entered a carriage return by mistake, you can end the API Provider reading loop by entering the following invalid command:
> *_trace: toto*
> *[<blank line>]*

If you make a syntax error in the API session window, an error reply is generated. You must enter a carriage return, signalling the end of the (erroneous) command and resumption of normal parsing. If you run the API Provider in the verbose mode, the following reminder message is displayed when an error is detected:

```
The rest of this API query will be ignored.
Make sure it is terminated (blank line) before entering
a new one.
```

If you enter a command that will take a long time to execute, the next command is not parsed until the current one is complete unless the API Provider has the asynchronous mode available and active.

Interactive sessions using input files

The keyboard mode causes an input file of commands to be executed, and enters interactive mode when those commands are completed. When the input file is completed, the API Provider waits for commands on its standard input as described in the message displayed when in the verbose mode:

```
Input source switched to standard input by -k flag.
```

The echo mode is especially useful when using a command input file. This mode causes each command to be displayed as a comment as it is executed.

You can create sieves and have their event notifications processed as part of a pipe. Use the daemon mode to force the API Provider to continue processing event notifications after the end-of-file sequence is encountered.

```
echo “<sieve creation query>”|\
<API Provider command line>|\
<filtering command line>
```

or

```
<API Provider command line>
-f <sieve creation query file>|\
<filtering command line>
```

The only way to terminate the Provider in this situation is to enter an abort sequence or kill the process as described in the message displayed when in the verbose mode:

```
API Provider waiting for Server Notifications (-d
flag).
Use ctrl-C or kill <process Id> to terminate.
```

Redirecting output

When piping the output from an interactive session into standard UNIX filtering utilities, remember that those programs usually buffer their output and can generally not be used to provide interactive output because the buffers are not displayed until the buffers are full or input terminates. Test your pipes and determine whether the output is as interactive as desired.

Ending an interactive session

To end an interactive session with the API Provider, use either the _end: command or an end-of-file (control-d).

The following _end: command is sent to the server, which shuts down the session.

```
_end:
<blank line>
```

The API Provider responds to either of these (_end: or control-d) with an _end: message with the USER_REQUEST keyword, and then terminates.

```
_end: USER_REQUEST
<blank line>
```

If the API User sends a syntactically invalid _end: message, such as one with a <reason> keyword, the API Provider should return a syntax error. Some APIs, however, will accept the message and ignore the keyword.

The interrupt (control-c) signal should not be used to terminate an API session. The control-c signal does not shut down the API Provider cleanly. The effect on any outstanding messages is unpredictable, and a proper _end: message may not be issued. The API User should therefore use either _end: or control-d to terminate cleanly.

If you abort a session using control-c while a command is being executed or while sieves are sending event notifications, an error reply and end reply is sent for each command and sieve.

```
_error: APPLICATION_ERROR API Provider interrupted by
user
_end: USER_REQUEST
<blank line>
```

API command line arguments

APIs are invoked as UNIX processes, and accept a number of command line arguments. The individual API Reference Guides describe the options available in each API and their use, but some common arguments are described in the table “API command line arguments” (page 74).

Table 8
API command line arguments

Argument	Function
-a	asynchronous mode: receives input and provides output as it occurs rather than in discrete blocks
-d	daemon mode: take API input from stdin or the file named by the -f option, then continue operating with no input (that is, only process events and any outstanding responses)

(Sheet 1 of 2)

Table 8 (continued)
API command line arguments

Argument	Function
-echo	echo API input to API output as API comments
-f <file>	take API input from file (default stdin)
-h <host>	specify the host to connect to for services (for example, the IPI server) (default localhost)
-help	give a summary of available command line arguments
-k	Keyboard mode: take API input from the file named by the -f option, then from stdin.
-o <file>	Send API output to file, overwriting (default stdout).
+o <file>	send API output to file, appending.
-v	Verbose mode: displays message to assist you. Used in interactive mode only.
-width <width>	Specify the line width for wrapping output with backslash (default is API-specific).
Note: For those arguments with an <option>, proper syntax includes a space between the argument and its option.	
(Sheet 2 of 2)	

Only the -help option is mandatory in any API Provider. Any error when processing the command line arguments results in a help message, as for -help.

API configured with INETD

If the API has been configured within INETD, your custom program need only make an IP call to the configured port. Refer to UNIX documentation for information on configuring IP ports within INETD.

Alternatively, you may wish to run the pserver program that performs much the same role as INETD but simplifies administration. For further information, see “pserver program” (page 76).

API and custom program on different platforms

If communication between the custom program and the Preside Multiservice Data Manager (MDM) workstation takes the form of the custom program logging on to the MDM workstation, then the custom program need only send the standard string for starting the API Provider. See the individual API Reference Guide for the startup string for a particular API Provider.

One way for a customer host to log on to an MDM workstation is to connect an RS-232-C port on the MDM workstation to an RS-232-C port on the customer host (using a null modem for short distances, two real modems for longer distances). The RS-232-C port on the MDM workstation must be enabled and configured to run getty (refer to UNIX documentation for details). The RS-232-C port on the customer host must be accessible to the customer application. The customer host documentation has the information for setting up and driving such a port.

API and custom program on the same workstation

If the custom program is to run on the Preside Multiservice Data Manager (MDM) workstation, the necessary piping can be set up from a separate program that starts up both the API and the custom program as follows:

```
couple <api startup command> <customer appl command>
```

The couple command

The couple command can be implemented as shown in the C source code example in /opt/MagellanNMS/cfg/macros/nms/src/couple.c.

API and custom program on different workstations

The simplest way to couple programs running on separate workstations is by using stream sockets. In this approach a client/server model is used to allow the programs to communicate. Two programs are required: one to act as a server (pserver) the other to act as a client (pcall). An additional program scall can be used to communicate interactively with a socket.

pserver program

To run an API as a network service and listen for incoming messages from another workstation on socket 4000, enter the following command:

```
pserver 4000 "<api startup cmd>" >>& api.4000.log &
```

When a call is received on socket 4000, the program accepts the call, and spawns <api startup command>, which connects stdin and stdout to the newly created stream or socket.

pserver lets you specify a socket number betwen 1024 and 5000.

pserver normally does not manage the processes it spawns. If pserver is terminated, the processes are allowed to continue. If you specify -e -m on the pserver command line (before the API startup command), pserver manages these processes and automatically terminates them if pserver is terminated in a normal manner (for example, from the Server Administration tool). With -e, you must also specify any argument to the API startup command as a separate argument to pserver. For example

```
pserver -m -e 4000 /opt/MagellanNMS/bin/gmdrapi -v \
>>& api.4000.log &
```

The pserver program can be implemented as shown in the C source code example in /opt/MagellanNMS/cfg/macros/nms/src/pserver.c. A compiled version of the pserver program exists in file /opt/MagellanNMS/bin/pserver. The compiled pserver program runs like a Preside Multiservice Data Manager (MDM) server that can be configured, edited, started and stopped with the MDM Server Manager Administration tool.

pcall program

To connect to an API as network service and send information to workstation bcars288 through stream socket 4000, enter the following command:

```
pcall bcars288 4000 <client application command>
```

The pcall program establishes a stream socket connection to port 4000 on workstation bcars288, and spawns the <client application command>, which connects stdin and stdout to the newly created stream socket. The client application communicates with the remote application through stdin and stdout.

The pcall program can be implemented as shown in the C source code example in /opt/MagellanNMS/cfg/macros/nms/src/pcall.c.

scall program

To try out remote access interactively to workstation bcars288 on stream socket 4000, enter the following command

```
scall bcars288 4000
```

The scall program can be implemented as shown in the C source code example in /opt/MagellanNMS/cfg/macros/nms/src/scall.c.

Appendix A
Message syntax summary

This appendix includes a set of tables that summarize the syntax and procedures of Preside Multiservice Data Manager (MDM) APIs. This appendix contains the following information:

- “Syntax and procedures of APIs” (page 79)

Syntax and procedures of APIs

“API messages” (page 41) has more details about the messages in this appendix. See the table “Message line labels” (page 79).

Table 9
Message line labels

Line label	Line purpose/contents	Into/Out of API Provider
`_action_type`	action type	I
`_attr`	attribute value	I/O
`_attr_id`	attribute name	I
`_block`	block construct	O
`_capability`	capability set	O
`_cmd`	command name	I
`_echo`	echo message	I
`_end`	end message	I/O
(Sheet 1 of 2)		

Table 9 (continued)
Message line labels

Line label	Line purpose/contents	Into/Out of API Provider
`_end_block`	end of block construct	O
`_end_resp`	end-of-response message	O
`_error`	error message	O
`_event_type`	event type	O
`_filter`	filter	I
`_inv_id`	invoke identifier	I/O
`_mod`	modification	I
`_obj_id`	object name	I/O
`_obj_class`	object class	I/O
`_password`	password	I
`_ref_obj_id`	reference object name	I
`_scope`	scope	I
`_sieve_id`	sieve identifier	O
`_sup_obj_id`	superior object name	I
`_time`	time	O
`_trace`	trace message	I
`_user_id`	user ID	I/O
`_version`	version message	O
(Sheet 2 of 2)		

Table 10
Message formats

Message type	Format
GET request	_cmd: GET [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value> [_scope: BASE | NEXT | ALL] [_filter: <attr name> <operator>[<attr value type> <attr value>]] … [_attr_id: <attr name> | ALL] …
GET response	(zero or more (one for each selected object), followed by an end of response message) [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value> [_attr: <attr name> <attr value type> <attr value>] …
SET request	_cmd: SET [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value> [_scope: BASE | NEXT | ALL] [_filter: <attr name> <operator>[<attr value type> <attr value>]] … _mod: <modify operator> <attr name> [<attr value type> <attr value>] …
SET response	(zero or more (one for each selected object), followed by an end of response message) [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value> [_attr: <attr name> <attr value type> <attr value>] …
(Sheet 1 of 5)	

Table 10 (continued)
Message formats

Message type	Format
ACTION request	_cmd: ACTION [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value> [_scope: BASE | NEXT | ALL] [_filter: <attr name> <operator>[<attr value type> <attr value>]] … _action_type: <action type> [_attr: <attr name> <attr value type> <attr value>] …
ACTION response	(zero or more (one for each selected object), followed by an end of response message) [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value> [_attr: <attr name> <attr value type> <attr value>] …
CREATE request	_cmd: CREATE [_inv_id: <invoke-id>] _obj_class: <object class> [_obj_id: <attr name> <attr value type> <attr value>] [_sup_obj_id: <attr name> <attr value type> <attr value>] [_ref_obj_id: <attr name> <attr value type> <attr value>] [_attr: <attr name> <attr value type> <attr value>] …
CREATE response	(one, followed by an end of response message) [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value>
(Sheet 2 of 5)	

Table 10 (continued)
Message formats

Message type	Format
DELETE request	_cmd: DELETE [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value> [_scope: BASE | NEXT | ALL] [_filter: <attr name> <operator>[<attr value type> <attr value>]] …
DELETE response	(zero or more (one for each selected object), followed by an end of response message) [_inv_id: <invoke-id>] _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value>
REGISTER request	_cmd: REGISTER [_inv_id: <invoke-id>] [_user_id: <user id>] [_password: <password>] [_attr: <attr name> <attr value type> <attr value>] …
REGISTER response	(one optional response, followed by an end of response message) [_inv_id: <invoke-id>] [_user_id: <user id>] [_capability: <capability set>] [_attr: <attr name> <attr value type> <attr value>] …
DEREGISTER request	_cmd: DEREGISTER [_inv_id: <invoke-id>]
DEREGISTER response	(one optional response, followed by an end of response message) [_inv_id: <invoke-id>]
(Sheet 3 of 5)	

Table 10 (continued)
Message formats

Message type	Format
EVENT-REPORT message	(sent by the API Provider, NOT responded to, NOT followed by an end of response message) _event_type: <event type> _sieve_id: <attr value> _obj_class: <object class> _obj_id: <attr name> <attr value type> <attr value> _time: <time> [_attr: <attr name> <attr value type> <attr value>] …
END-OF-RESPONSE message	(sent after zero or more responses to a request, the <command name> must be included unless it cannot be determined from the request) [_inv_id: <invoke-id>] _end_resp: [<command name>] _time: <time>
ERROR message	(sent in place of a response; followed by zero or more responses and an end-of-response) [_inv_id: <invoke-id>] _error: <error name> [<error title>] [_attr: <attr name> <attr value type> <attr value>] …
VERSION message	(sent once by the API Provider at startup) _version: <version number> <version string>
END request	_end: <blank line>
END response	_end: USER_REQUEST (sent back to the API User from the API Provider as a response to an END request)
(Sheet 4 of 5)	

Table 10 (continued)
Message formats

Message type	Format
END message	(sent once by the API Provider to terminate the API session) _end: FATAL_ERROR _end: SYSTEM_SHUTDOWN _end: LOST_CONNECTION _end: LOST_SESSION
TRACE message	(sent by the API User, NOT responded to) _trace: ON | OFF
ECHO message	(sent by the API User, NOT responded to) _echo: ON | OFF
BLOCK construct	(sent in place of a single message line to allow multi-line data values) _block: <line label> [<parameter value>] … <data value> … _end_block:
(Sheet 5 of 5)	

Table 11
Message fields

Field	Syntax	Description
<action type>	String	The name of an action
<attr name>	String	The name of an attribute
<attr value type>	String: B | I | H | D | S | FS | NI | LI | E | SS | SI | SB | RS | RI | RD	The data type of the attribute value that follows
<attr value>	Value	Attribute value, formatted as defined for the preceding <attr value type>
<capability set>	String	The capability set assigned for the API session
(Sheet 1 of 3)		

Table 11 (continued)
Message fields

Field	Syntax	Description
<command name>	String: GET | SET | CREATE | DELETE | ACTION | REGISTER | DEREGISTER	The type of command that is being closed out with an end-of-response message
<data value>	String	An attribute value that may span multiple lines in a block message
<error title>	String	A title or textual name for the error, suitable for displaying
<error name>	String	The kind of error that is being reported See the table “Error names” (page 87) for the list of legal error names and their meanings.
<event type>	String	The name of an event
<invoke-id>	Integer	A unique number for each request
<line label>	String	A line label, taken from the table “Message line labels” (page 79)
<modify operator>	String: ADD | DEL | DEF | REP	The operator to apply to the attribute in a SET message
<object class>	String	The class of the base or selected object
<operator>	String: P | EQ | NE | LT | GT | GE | LE | LEFT | MIDDLE | RIGHT | IN | NOTIN	The kind of comparison/test to be applied in a filter
<parameter value>	String	A parameter value in a _block line, formatted according to the message type specified by the <line label>
<password>	String	The password of the API User
(Sheet 2 of 3)		

Table 11 (continued)
Message fields

Field	Syntax	Description
<time>	String	Date and time, encoded as: yyyy mm dd hh mm ss th where yyyy is the year, mm is the month (01-12), dd is the day (01-31), hh is the hour (00-23), mm is the minute (00-59), ss is the second (00-59), t is tenths of a second (0-9) and h is hundredths of a second (0-9). The tenths and hundredths are optional, and some API Providers may ignore them on input. When an API Provider that supports the th field compares two D values, one with the th specified, and one without, the missing th is assumed to be 00.
<user id>	String	The user ID of the API User
<version number>	String: x.y format	A string giving the version number of the API
<version string>	String	A string describing the API Provider
(Sheet 3 of 3)		

Table 12
Error names

Error name	Description	Command support
ACCESS_DENIED	The request exceeds the assigned privileges for the API User.	Supports GET, SET, ACTION, CREATE, DELETE.
APPLICATION_ERROR	Unspecified error. The _attr lines further qualify the error.	Supports GET, SET, ACTION, CREATE, DELETE, REGISTER, DEREGISTER.
(Sheet 1 of 3)		

Table 12 (continued)
Error names

Error name	Description	Command support
UNRECOVERABLE_ERROR	An error from which recovery is not possible, for example: out of memory errors, fork failures, internal inconsistencies, failure to connect to a server, failure to open a necessary file.	
DUPLICATE_OBJECT	The object to be created already exists.	Supports CREATE.
INIT_ERROR	An error occurred during API Provider initialization. The API version number is the first token in <error title>.	Not applicable.
INVALID_ACTION_TYPE	The action type is not recognized.	Supports ACTION.
INVALID_ATTRIBUTE_NAME	An attribute name in the request is invalid.	Supports SET, ACTION, CREATE, REGISTER.
INVALID_ATTRIBUTE_VALUE	An attribute value in the request is invalid.	Supports SET, ACTION, CREATE, REGISTER.
INVALID_FILTER	A filter is invalid or inappropriate for the attribute that is being tested.	Supports GET, SET, ACTION, DELETE.
INVALID_OBJECT_ID	The object name in the _obj_id line is invalid.	Supports GET, SET, ACTION, CREATE, DELETE.
INVALID_OBJECT_CLASS	The object class in the _obj_class line is invalid.	Supports GET, SET, ACTION, CREATE, DELETE.
INVALID_SCOPE	The scope is invalid.	Supports GET, SET, ACTION, DELETE.
LOGON_FAILS	The user ID or password is invalid.	Supports REGISTER.
MISSING_ATTRIBUTE_VALUE	No default value is available for an attribute, and the CREATE request did not supply one.	Supports ACTION, CREATE, REGISTER.
(Sheet 2 of 3)		

Table 12 (continued)
Error names

Error name	Description	Command support
MODIFICATION_ERROR	A SET request attempted an illegal modification (for example, an ADD to a single-valued attribute).	Supports SET.
NO_SUCH_OBJECT_ID	The object name in the _obj_id line is valid, but no such object exists.	Supports GET, SET, ACTION, CREATE, DELETE.
OUT_OF_SEQUENCE	A message was sent out of sequence (for example, a REGISTER was required first).	Supports GET, SET, ACTION, CREATE, DELETE, REGISTER, DEREGISTER.
SYNTAX_ERROR	The message was improperly formatted (for example, any error in parsing a received message, if the error can be detected without any knowledge of the object model).	Not applicable.
NOT_SUPPORTED	A message line or keyword was recognized/parsed, but is not supported.	Not applicable.
(Sheet 3 of 3)		

Index

Preside Multiservice Data Manager

Application Programming Interface

Primer

R14.3

Publication: 241-6001-200
Document status: Standard
Document version: 14.3RSUP
Document date: December 2003
Printed in Canada