NORTEL NETWORKS

Preside Multiservice Data Manager

Embedded Programming Interface

Reference Guide

241-6001-211

Preside Multiservice Data Manager

Embedded Programming Interface

Reference Guide

Publication: 241-6001-211
Document status: Standard
Document version: 14.3RSUP
Document date: December 2003

Printed in Canada

Publication history

December 2003

14.3RSUP 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 those who write applications that interact with multiple Preside Multiservice Data Manager (MDM) interfaces to customize and automate network management activities. The EPI enhances the usability of MDM Application Programming Interfaces (APIs) and other MDM customization utilities.

What you need to know

This document assumes you have knowledge in the following areas:

- how to log on to a Preside Multiservice Data Manager (MDM) workstation
- your network model
- UNIX operating system
- UNIX C shell
- UNIX text editor (such as vi)

- DeskTop Korn shell (DtKsh), Tool Command Language (Tcl), Perl, or C/C++.

How this document is organized

241-6001-211 *Preside MDM Embedded Programming Interface Reference Guide* contains the following sections:

- "Introducing EPIs" (page 19) describes the Preside Multiservice Data Manager (MDM) EPI.
- "DtKsh Embedded Programming Interface" (page 27) describes the MDM EPI in the DtKsh scripting language.
- "Tcl Embedded Programming Interface" (page 137) describes the MDM EPI in the Tcl scripting language.
- "Perl Embedded Programming Interface" (page 245) describes the MDM EPI in the Perl scripting language.
- "C/C++ Embedded Programming Interface" (page 349) describes the MDM EPI accessible from C/C++ code.

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 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:

- 241-6001-100 *Preside MDM Installer Guide*
- 241-6001-105 *Preside MDM Software Packaging Reference Guide*
- 241-6001-200 *Preside MDM Application Programming Interface Primer*
- 241-6001-201 *Preside MDM Network Model API Reference Guide*
- 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*
- 241-6001-303 *Preside MDM Administrator Guide*
- Libes, D., *Exploring Expect*, O'Reilly & Associates, 1995
- Ousterhout, J.K., *Tcl and the Tk Toolkit*, Addison-Wesley, 1994
- Pendergrast, J.S. Jr., *Desktop KornShell Graphical Programming*, Addison-Wesley, 1995
- Wall, Larry, Tom Christiansen, and Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, 1996

Chapter 1
Introducing EPIs

This section introduces the Embedded Programming Interfaces (EPI) and describes its purpose. This section contains the following information:

Embedded programming interfaces

The Preside Multiservice Data Manager (MDM) EPIs provide powerful and efficient access to the MDM Application Programming Interface (API) and Command Access programming interfaces for network operations automation. It lets you write applications to collect data from, and interact with, multiple MDM interfaces at the same time. For example, you can correlate multiple alarm streams. You can also send commands to network elements that are triggered by Network Model state changes or other notifications. EPIs make it easier to perform complex API query sequences where some queries depend on the results of previous ones (for example, a recursive descent down parts of the Network Model).

MDM provides several interfaces and utilities that allow you to customize and automate network management activities and MDM operations and tools. These include the following

- **MDM Application Programing Interfaces (APIs)**
  APIs allow you to extract and manipulate management data used by MDM servers and tools. APIs are ASCII query languages that are based on CMISE/CMIP concepts.

- **Command Access (cmccmd)**
  The cmccmd utility allows you to write UNIX-based macros for on-switch commands, including SNMP command processor (snmpCmd) macros.
- **Customer Database**
  The Customer Database allows you to associate textual information with specific component names. The Customer Information Database is managed by the CDB Server and is available through the Customer Database tool.
- **Real Time Alarm Collection (RTAC)**
  RTAC spools alarms from the GMDR surveillance server to workstation files. you can scan the contents of the spooled files to produce historical alarm reports.
- **Network Reporting System (NRS)**
  NRS lets you collect configuration data from DPN and Passport network elements and produce network-wide reports.
- **Programing utilities**
  MDM supports a number of programming utilities that allow you to interface with MDM and manipulate the data it contains (for example, providing access to the MDM Context Server to identify Service Selection and Hot Context settings, manipulating MDM component IDs, and pattern matching).

EPI enhances the usability of these interfaces and utilities by providing access to them through the DeskTop Korn Shell (DtKsh), Tool Command Language (Tcl), and Perl scripting languages. There is a C/C++ version of the EPI interfaces for your coded applications. EPIs are also supported through a CORBA IDL interface for remotability and language independence. EPI supports the following releases: DtKsh (CDE 1.0.x and later), Tcl7.5 and later (and specializations like Tk and Expect), and Perl version 5.0003 or later. The C/C++ version has been tested using Sun's SunPro C and C++ compilers. The CORBA IDL servants are supported on Orbix version 3.3.1.

Tcl, DtKsh, and Perl provide a powerful scripting environment. EPI scripts are very efficient since they don't need to spawn multiple processes. This alleviates the condition where one process is needed for each API provider,

one process for cmccmd calls, and many processes for parsing the output provided by these interfaces. As well, since the connection to the API Session servers is made only once, there is a reduction in the IPC connection cost.

The C/C++ interfaces can be used when more performance is required (especially in the code surrounding the EPI routines) or when you need to interact with systems whose interfaces are only available in traditional compiled programming languages.

The CORBA EPI interfaces can be used when the client application cannot reside on the same workstation as the MDM system itself or the client's implementation language is not one of those named above (e.g. Java). Note that MDM does not distribute the necessary Orbix daemon on the server side which will have to be purchased and installed separately. MDM provides the IDLs and the necessary servant applications.

> ***Note:*** The CORBA EPI does not currently support all the capabilities of the other EPI interfaces. In this release, only the Command and API interfaces are supported. Other interfaces will be rolled out in future releases.

> ***Note:*** EPIs do not change the API or command syntax. The API language used for queries and replies remains the same. Also, the command reply syntax remains governed by the network elements (in addition to the destination name prefix already required by cmccmd). The only impact of EPI is that the information from the interfaces is provided in a more efficient and pre-parsed manner. EPIs do not replace APIs. Many tasks may still be implemented as efficiently using APIs (for example, when a single streamlined query is required).

The MDM EPI provides programming access to shared libraries for

- Generic API Access
  - — connecting to or disconnecting from a generic API server (Alarm, Network Model, Host Group Directory Service)
  - — sending API queries (in ASCII form)
  - — receiving/skipping API replies (in pre-tokenized ASCII form) (synchronous and asynchronous)

  - — extracting/searching replied API records and fields
  - — specialized routines for Alarm&Status, Network Model and Host Group Directory access
- Command Access
  - — connecting to or disconnecting from the Command Session servers
  - — connecting to or disconnecting from an OA/Passport Group
  - — sending commands
  - — querying a list of available OAs and Passport Groups
  - — receiving replies, whole or one line at a time (synchronous and asynchronous)
  - — receiving Passport pre-parsed Passport display command requests
  - — parsing replies (pattern matching and field extraction)
- Customer Database
  - — connecting to or disconnecting from the CDB Servers
  - — fetching the information associated with a component name
  - — querying the database with patterns for the component name, data, or related component name (synchronous and asynchronous)
  - — receiving query replies
  - — storing (adding or replacing) information associated with a component name
  - — erasing the database entry associated with a component name
- Real Time Alarm Collection
  - — querying the spooled data between two date boundaries
  - — specifying pattern-based filters on any alarm attribute
  - — retrieving query replies (alarms) and their attributes
- Network Reporting System
  - — querying the collected data by node and configuration name patterns, configuration file names, configuration version (keyed, dated, or latest)

  - — specifying the component types to be reported
  - — retrieving query replies (components) and their attributes
  - — loading and querying the NRS configuration data schema (RDFs)
- Miscellaneous
  - — manipulating MDM component names
  - — accessing MDM contexts
  - — group style pattern matching
  - — long integer arithmetic
  - — setting timer callbacks

With this added functionality, you can use EPIs as the basis for more network operations automation. You can use EPIs to create scripts that react to simple and complex sets of management events. This is accomplished by sending correcting commands to the network elements or by tracking the event through logging or alarm injection. You can also use EPIs to help integrate management data from MDM into umbrella management systems.

Delivery

The Preside Multiservice Data Manager (MDM) EPI is delivered with the MDM Base software in three additional shared libraries and two inclusion modules:

- **`/opt/MagellanNMS/lib/libEPI.so:`**
  contains the base EPI (including base, API, Command, Customer Database, and Miscellaneous interfaces) support and references to the appropriate MDM libraries.
- **`/opt/MagellanNMS/lib/libEPIR.so:`**
  does similarly for the RTAC and NRS interfaces.
- **`/opt/MagellanNMS/lib/libEPIKsh.so`**
  contains the DtKsh EPI support code for the API, Command, Customer Database, and Miscellaneous interfaces and a reference to `libEPI.so`.
- **`/opt/MagellanNMS/lib/libEPIKshR.so`**
  does the same for the RTAC and NRS interfaces.

- **`/opt/MagellanNMS/lib/libEPITcl.so`**
  contains the Tcl EPI support code for the API, Command, Customer Database, and Miscellaneous interfaces and a reference to `libEPI.so`.
- **`/opt/MagellanNMS/lib/libEPITclR.so`**
  does the same for the RTAC and NRS interfaces.
- **`/opt/MagellanNMS/lib/libEPIPublic.so`**
  contains the C/C++ EPI support code for the API, Command, Customer Database, and Miscellaneous interfaces and a reference to `libEPI.so`.
- **`/opt/MagellanNMS/lib/libEPIPublicR.so`**
  does the same for the RTAC and NES interfaces.
- **`/opt/MagellanNMS/lib/nmsepi.ksh`**
  is a Ksh source file that loads the EPI built-in routines for the API, Command, Customer Database, and Miscellaneous interfaces into a Korn Shell script (KSH 93 or CDE DeskTop Korn Shell)
- **`/opt/MagellanNMS/lib/nmsepir.ksh`**
  does the same for the RTAC and NRS interfaces
- **`/opt/MagellanNMS/lib/nmsepi.tcl`**
  is a Tcl source file that loads the EPI built-in routines for the API, Command, Customer Database, and Miscellaneous interfaces for Tcl-based scripts (Tclsh or derived Tcl Shells, version 7.5 and later)
- **`/opt/MagellanNMS/lib/nmsepir.tcl`**
  does the same for the RTAC and NRS interfaces
- **`/opt/MagellanNMS/lib/EPI.pm`**
  is a Perl 5 module that loads the EPI built-in routines for the API, Command, Customer Database, and Miscellaneous interfaces for Perl-based scripts
- **`/opt/MagellanNMS/lib/EPIR.pm`**
  does the same for the RTAC and NRS interfaces
- **`/opt/MagellanNMS/lib/EPIPublic.h`**
  `/opt/MagellanNMS/lib/EPIPublic.hxx`
  are the C/C++ include files exporting the public EPI library types and functions for the API, Command, Customer Database, and Miscellaneous interfaces.

- **`/opt/MagellanNMS/lib/EPIPublicR.h`**
  `/opt/MagellanNMS/lib/EPIPublicR.hxx`
  do the same for the RTAC and NRS interfaces.
- **`/opt/MagellanNMS/lib/idl/EPIBase.idl`**
  `/opt/MagellanNMS/lib/idl/EPICmd.idl`
  `/opt/MagellanNMS/lib/idl/EPICmdCallback.idl`
  `/opt/MagellanNMS/lib/idl/EPIContextCallback.idl`
  `/opt/MagellanNMS/lib/idl/EPIAPI.idl`
  `/opt/MagellanNMS/lib/idl/EPIAPICallback.idl`
  these are the CORBA IDLs for the CORBA EPI interface.
- **`/opt/MagellanNMS/lib/idl/cplusplus/EPIAPI.hxx`**
  `/opt/MagellanNMS/lib/idl/cplusplus/EPIAPICallback.hxx`
  `/opt/MagellanNMS/lib/idl/cplusplus/EPIBase.hxx`
  `/opt/MagellanNMS/lib/idl/cplusplus/EPICmd.hxx`
  `/opt/MagellanNMS/lib/idl/cplusplus/EPICmdCallback.hxx`
  `/opt/MagellanNMS/lib/idl/cplusplus/`
  `EPIContextCallback.hxx`
  these are the pre-compiled C++ skeleton interfaces for the corresponding CORBA IDLs.

  Note: Since each library refers to its depending libraries, only the required library (for example, libEPIKsh.so or libEPITcl.so) must be loaded; the dependent libraries are automatically loaded.

To use MDM EPI routines in a DtKsh, Tcl, or Perl script, you need to source any corresponding inclusion modules (for example, nmsepi.ksh, nmsepir.ksh, nmsepi.tcl, or nmsepir.tcl) with the appropriate language construct (‘.’ in DtKsh, source in Tcl), or include any appropriate EPI Perl modules (use the Perl command).

To use the C/C++ EPI routines, you need to compile your program with one or more of the EPIPublic.h, EPIPublicR.h, EPIPublic.hxx, or EPIPublicR.hxx include files and link it with the libEPIPublic.so or libEPIPublicR.so, or both, shared libraries. You must also link with the Xt/X11 shared libraries to have access to its event loop functionality.

To use the CORBA EPI interface, you need to take the required IDLs and compile language specific stubs to use in your client implementation (in the usual manner dictated by the implementation language's CORBA interface capabilities). The servant processes for the EPI interfaces you intend to use must then be registered into the Orbix object server on the MDM machine side. The CORBA client can then locate the particular servant it wants (both Orbix BIND and standard Naming Service identifications are supported) and start issuing calls to it. The CORBA EPI servants are configured with a per-client spawning discipline (each client process automatically gets its own private servant process). If the client process terminates accidentally, the servant will automatically terminate and release any workstation or network resources it is holding. The servants do not support any form of persistence so all context is lost between invocations.

Chapter 2
DtKsh Embedded Programming Interface

This section describes the Embedded Programming Interface (EPI) in the DeskTop Korn Shell (DtKsh) scripting language and contains the following information:

Code conventions

There are two code conventions used in this chapter:

- \
  A back slash (\) indicates that the line of code continues on the next line space.
- #
  A message line that starts with an octothorp (#) is treated as a comment.

Integration methodology

DtKsh scripts are executed by the dtksh (*/usr/dt/bin/dtksh* in the Solaris CDE Shell interpreter.

DtKsh, and more generally Korn Shell93, use a simple extension technique. The only requirement is to invoke the following command in the script:

```
builtin -f <shared library path> <built-in function \
bindings list...>
```

where:

`<shared library path>` is the integration library.

`<built-in function bindings list...>` is the list of names for the new built-in functions to be extracted from this library.

The built-in command call is performed by an integration module. The DtKsh EPI extensions are divided into two sets. To access the base, API, Command, Customer Database and Miscelaneous interfaces, DtKsh EPI script must load the integration module as follows:

```
. /opt/MagellanNMS/lib/nmsepi.ksh
```

To access the RTAC and NRS interfaces, the script must load the integration modules as follows:

```
. /opt/MagellanNMS/lib/nmsepir.ksh
```

where:

`. (dot)` is the source command.

Both sets of interfaces can be loaded by sourcing the two files.

Interface

DtKsh EPI provides the following groups of built-in commands:

- **Base**
  Provides mapping to the EPI base routines and utilities as well as additional housekeeping routines.
- **Generic API Access**
  Provides access to the Generic Application Programming Interface (API) commands and replies.
- **Specialized API Access**
  Provides specialized and value-added access to specific API Providers such as the Alarm and Status, Network Model, and Host Group Directory Service (HGDS) APIs.
- **Command Access**
  Provides access to the Preside Multiservice Data Manager (MDM) command macro capabilities.
- **Customer Database Access**
  Provides access to the MDM Customer Database capabilities.
- **Real-Time Alarm Collection (RTAC)**
  Provides access to the spooled alarms collected by RTAC.
- **Network Reportinng System (NRS)**
  Provides access to the network-wide configutation data collected by NRS.

Command usage information

The following information applies to built-in commands:

- **help**
  All commands support a -h (help) option that displays arguments to a command
- **command argument**
  You need to specify all command arguments in the indicated order.

- **command output**
  Some commands support a -out argument that prints the command's output to the standard output stream. For information on using UNIX pipes with DtKsh EPI commands, see "Output redirection and piping" (page 134).

- **unnamed and named connections**
  By default, the connection-oriented commands (API or Command Access) operate on a single unnamed connection. It is possible, however, to create concurrent named connections. To do so, specify the connection name as an argument (`-iname`) to a command (except for the `NMSEPIDefaultAPI`, `NMSEPIDefaultCmd`, `NMSEPIDefaultCdb`, `NMSEPIDefaultRTAC`, or `NMSEPIDefaultNRS` commands).

- **error output**
  Error messages are usually output to the standard error stream. You can stop output to the standard error stream by using the NMSEPISet command.

- **return codes**
  Commands return (as exit codes) one of the following values:

0	success
1	error in argument list
2	default/named interface not created/initialized
3	default/named interface not connected to server
4	operation failed
5	operation timed out
6	no more replies
7	attempt to create an exiting interface (name)

  These return codes are available through the $? shell variable and the `NMSEPI_RESULT` environment variable. These return codes also allow you to use the commands directly in an *if* clause.

Example

```
if NMSGMDRAPIConnect -ssel
then
... # successfully connected
fi
```

Base

In general, the first task in writing DtKsh EPI scripts is to initialize an interface. There are special purpose commands to do this, but the Base also provides a general routine:

- **NMSEPIInit**
  Initializes the EPI environment. If DtKsh is to be used to implement a graphical user interface, you must invoke the XtInitialize command before using `NMSAPInit` (or its specific API command version: `NMSAPIInit, NMSGMDRAPIInit, NMSNMAPIInit, NMSHGDSAPIInit, NMSCmdInit,` or `NMSCdbInit`).

- **NMSEPITerm**
  Drops all current API and command interfaces, and terminates the IPC environment. This is sometimes required when another EPI script (which makes and maintains its own API or command interfaces) is reused and invoked in the same UNIX process. The second script would otherwise fail because of duplicate server connections.

EPI normally outputs error messages (for example, if the connection to a server fails to be created) on the standard error stream. You can prevent this from occurring by using the following command:

- **NMSEPISet +err|-err**
  Controls whether error messages are output (`+err`) or not output (`-err`) to the standard error stream.

API and Command Access commands usually apply to a default connection (a single connection for all APIs, another one for Command Access, and another one for Customer Database access). When using multiple concurrent

connections, you can use the following commands to make a specific command the default and thereby avoid having to specify its name as an argument in every command:

- **NMSEPIDefaultAPI [-iname <name>]**
  NMSEPIDefaultCmd [-iname <name>]
  NMSEPIDefaultCdb [-iname <name>]
  NMSEPIDefaultRTAC [-iname <name>]
  NMSEPIDefaultNRS [-iname <name>]
  If it exists, makes the named interface connection the default. If no interface is named, the original unnamed default interface, if any, is set to be the default. The last two calls are available only with the EPI reporting extension (RTAC and NRS interfaces).

  Note: There is only one default API interface for all API types (for example, Alarm and Status, Network Model).

The following Preside Multiservice Data Manager (MDM) utilities are supported:

- **NMSEPIConvertCompId [-out]**
  [-canon|-disp|-type|-mnem|-ep1|
  -ep2|-dpn|-switch] <component ID>
  Converts the specified component ID and returns the result in the `NMSEPI_COMP_ID` environment variable and standard output (if `-out` is used). The conversions are as follows:

  `-canon`
  converts to canonical API format (for example, PM AM1 PE 1 PI 1).

  `-disp`
  converts to display format (for example, PM/AM1 PE/1 PI/1).

  `-type`
  extracts the module/link type (for example, PM or NL).

  `-mnem`
  extracts the module mnemonic (for example, AM1).

  `-ep1` and `-ep2`

extract the first and second link endpoints in canonical API format.

`-dpn`
converts a DPN-100 OA or a PE/PI/PO component ID to a form suitable for commands (<mnemonic> [pe <pe#>|<pi #> [<po #>]]). (For example, AM1 11.)

`-switch`
returns the module-level component ID in canonical API format (for example, PM AM1).

`-omni`
returns the Preside Multiservice Data Manager (MDM) HP-OpenView DeskTop compatible component ID (similar to display format except for link names).

Examples

```
NMSEPIConvertCompId -ep1 "$myLinkId"
echo "Endpoint1: $NMSEPI_COMP_ID"

NMSEPIConvertCompId -dpn "$myDPNPort"
NMSCmdSendCommand $myOA "$NMSEPI_COMP_ID enable"
```

- **NMSEPICompareCompIds <component ID1> <component ID2>**
  Compares the two returning component IDs and returns (instead of the usual return codes) 0 if they are identical, <0 if the first component ID precedes the second, and >0 if the second precedes the first.

  Note: Since UNIX shells do not support negative numbers as return codes, use the `NMSEPI_RESULT` environment variable to determine the component ordering.

- **NMSEPIConvertTime [-out] [-stc] -api|-tostc|-epoch|-unix|-alarm|-pp |-ftime <format>|-offset <nb> [days] [<time string>]**
  Converts the specified time string and returns the result in the `NMSEPI_OUTPUT_TIME` environment variable and standard output (if `-out` is used). The input time can be in the following formats:

API format (`YYYY MM DD HH MM SS`)
Common Alarm format (`YY-MM-DD HH:MM:SS`)
UNIX Epoch (`<number of seconds since 1970>`)
Passport reply format (`YYYY-MM-DD HH:MM_SS`)

The returned time is in the same time frame as the input one. However, exceptions occur if you specify `-stc` or if you select the `-tostc` conversion. If you specify `-stc` the input time is assumed to be in Standard Time Coordinates, that is, Greenwich Mean Time (GMT). If you use `-stc` and not `-tostc,` then the output time converts from STC to local workstation time. If you do not specify the input time , the current workstation time is used and `-stc` is ignored.

The conversions are as follows:

`-api`
produces the time in API format.

`-tostc`
produces the time in API format converted to Standard Time Coordinates.

`-epoch`
returns the time as a UNIX Epoch value (number of seconds since 1970).

`-unix`
returns the time as the default time format for the workstation's LOCALE (see `man -s3c ctime`).

`-alarm`
returns the time in Common Alarm format.

`-pp`
returns the time in Passport reply format

`-ftime <format>`
returns the time in the specified format (see `man -s3c strftime`, 100 characters maximum).

`-offset <nb> [days]`
if `days` are not specified, returns the time in API format after applying

the specified positive or negative offset in seconds, otherwise returns the days

`-secsinday`
returns the number of seconds from the previous midnight and the specified time (or the current time).

`-stcoffset`
returns the number of seconds between the local workstation time and the coordinated universal time (UTC) and, if needed, taking daylight savings time into consideration. The offset is positive going west from UTC (same as UNIX timezone/altzone).

Example

```
NMSEPIConvertTime -offset -7 days
echo "$NMSEPI_OUTPUT_TIME"
```

returns the time 7 days ago, for example:

```
2000 03 24 17 01 19
```

```
NMSEPIConvertTime -epoch "2000 03 24 17 01 19"
echo "$NMSEPI_OUTPUT_TIME"
```

returns the same as UNIX epoch, for example:

```
953935279
```

```
NMSEPIConvertTime -ftime "Time was: %a %b %d %l:%M%p" \
                  "953935279"
echo "$NMSEPI_OUTPUT_TIME"
```

returns the same information using a custom format:

```
Time was: Fri Mar 24  5:01PM
```

- **NMSEPIPatternMatch [-out] [-g] <pattern> <target> [<substitute>]**

  Performs pattern matching of <pattern> with <target>. Specify <pattern> with grep syntax. You can include one `\(\)` delimited sub-pattern, which is returned if there is a match instead of the full matched portion. If you specify <substitute>, the matching sub-string is replaced by <substitute>. The `-g` option substitutes all matching sub-strings. The matched or substituted string is returned in the `NMSEPI_OUTPUT_MATCH` environment variable (and standard output if `-out` is used).

Example

```
if NMSEPIPatternMatch "pe \([0-9]*\)down" \
                "$NMSEPI_OUTPUT_TEXT"
then
   echo "PE# $NMSEPI_OUTPUT_MATCH is down."
...
```

- **NMSEPIGetContext [-out] [-ssel|-user|-ws] <var. name>**
  NMSEPISetContext -user|-ws <var. name> <value
  Gets or sets an Preside Multiservice Data Manager (MDM) context variable. The context corresponds to the Service Selection for -ssel, the current User Session for `-user`, and the Workstation Wide for `-ws`. `NMSEPIGetContext` sets the `NMSEPI_CTX_VALUE` environment variable to the extracted value (and outputs it to standard output if -out is used).

  Example

  ```
  NMSEPIGetContext -user "DPN_QUICK_STEP"
  echo "Hot context: $NMSEPI_CTX_VALUE"
  ```

- **NMSEPIRegisterContextInterest [-user|-ws <callback> <var. names...>]**
  Binds the specified DtKsh command string, `<callback>`, (typically a function invocation and its arguments) to the specified Preside Multiservice Data Manager (MDM) Context variables and domain (`-user` for the User Session and `-ws` for Workstation Wide). This command string will be executed whenever one of the named variable changes value and the script is in an event loop (`NMSEPIEventLoop` or `XtMainLoop`). The following environment variables are available to the callback:

  `NMSEPI_CB_REASON`
  is set to `CONTEXT`.

  `NMSEPI_CTX_VAR`
  is set to the the named of the changed variable.

  `NMSEPI_CTX_VALUE`
  is set to the new value for the variable.

Only one such callback can be registered. To deregister the callback, call this command without any arguments.

Example

```
function componentHotContext {
    # react to the hot context selection
    # whose component name value is in
    # $NMSEPI_CTX_VALUE
    ...
}
...
NMSEPIRegisterContextInterest -user \
    "componentHotContext" "DPN_QUICK_STEP"
...
NMSEPIEventLoop
```

- **NMSEPIEventLoop**
  Initiates an Xt-based event loop for asynchronous message handling. This command never returns. The processing scripts are then performed by the callback DtKsh code that is bound to the API and command interfaces (see `NMSEPIRegisterContextInterest`, `NMSAPIBindCallback`, `NMSCmdBindCallback`, and `NMSCdbBindCallback`). `NMSEPIEventLoop` is equivalent to the `XtMainLoop` command of DtKsh.

- **NMSEPITimer set <msec> <callback>**
  | clear <timer name>
  The first form, `set`, of this command binds the specified DtKsh command string, `<callback>`, (typically a function invocation and its arguments) to be invoked once in `<msec>` milli-seconds when the script is in an event loop (`NMSEPIEventLoop` or `XtMainLoop`). The created timer is given a name returned as the value of the `NMSEPI_TIMER_NAME` environment variable. This name can be used in the second form, `clear`, to cancel the timer. The callback code is invoked with the following environment variables:

  `NMSEPI_CB_REASON`
  is set to `TIMER`.

NMSEPI_TIMER_NAME
is set to the name of the timer that has expired.

Example

```
function rescanList {
    ...
    # re-create the timer to be called again
    NMSEPITimer set $(( 5 * 60 * 1000 )) “rescanList”
}
...
NMSEPITimer set $(( 5 * 60 * 1000 )) “rescanList”
...
NMSEPIEventLoop
```

- **NMSEPILongArith [-out] [-u] <integer value> +|-|x|/|% <integer value>**

  Performs long integer arithmetic on the values provided (`+` addition, `-` subtraction, `x` multiplication (`- - *` is also supported but must be escaped in DtKsh), `/` division, and `%` modulo). If `-u` is used, unsigned arithmetic is performed. The results are returned as a string in the `NMSEPI_RESULT` environment variable and on the standard output stream if `-out` is specified. This is useful for Passport command output where you specify some values as long integers.

  Example

```
NMSEPILongArith $currByteCount - $prevByteCount
echo “Delta: $NMSEPI_RESULT”
```

Generic API access

These commands provide access to the Generic API at the same level as if you were using the Generic API Provider utility, *genapi*, directly. At this level, no special knowledge of the individual API types is required; all queries can be handled in a generic way. (For more information, see 241-6001-200 *Preside MDM Application Programming Interface Primer*.

Generic API Access commands are used in the following sequence:

1 Initialize an API interface.

2 Connect to the API server.

3 If the server requires it, send a `REGISTER` message and wait for its reply.

4 Send an API query.

5 Receive the next reply and extract the needed information.

6 Disconnect from the API server.

7 Drop the API interface.

Multiple queries can be issued, but individual API interfaces will handle these queries synchronously, one at a time (except for sieve event notifications). It is possible, however, to create multiple-named API interfaces (each one resulting in its own connection to the server and therefore capable of independent, parallel queries). Note also that a single API interface can be reused (disconnected and reconnected to another server).

The same command sequence is used by the Specific API Access areas, but specialized commands are sometimes added to handle the particular details of an API type. For more information on specialized commands, see "Specialized API access" (page 45).

Generic API Access commands

The following Generic API Access commands are provided:

- **NMSAPIInit [-iname <name>] <API dictionary path>**
  Initializes a Generic API interface using the indicated API dictionary. If more than one interface is to be used, you should provide a name for it. This name must be indicated with the `-iname` option on all commands targetted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

  Note: Specialized API Access provides its own version of this command, which hides the detail of the API dictionary path. For more information, see "Specialized API access" (page 45).

- **NMSAPIDrop -iname <name>**
  Drops the named API interface to clear up the connection or so it can be reused.

- **NMSAPIConnect [-iname <name>] <service name> [<host>]**
  Connects the default or named interface to the specified API server, optionally on a different Preside Multiservice Data Manager (MDM) host.

Example

```
NMSAPIInit /opt/MagellanNMS/lib/api/GMDR.dict
if NMSAPIConnect GMDR localhost
then
  ... # connection successful
```

- **NMSAPIDisconnect [-iname <name>]**
  Disconnects the default or named interface from its current API server.

- **NMSAPIRegister [-iname <name>] <user id> [<password>] [-attr <attribute lines>]**
  Sends a `REGISTER` message through the default or named interface with the specified parameters

  where:
  `-attr <API attribute lines>` is one or more API attribute lines to add to the message (for example, the `"_attr: userCapability E mdInject"` line needed to register to IMDR with alarm injection capabilities).

  This command is actually a combination call, since it will both send the query and wait for the reply. The return code therefore indicates the success or failure of the complete `REGISTER` command-reply sequence.

  Example

```
if NMSAPIRegister toto \
     -attr "_attr: userCapability E mdInject"
then
... # successful register for alarm injection
```

- **NMSAPISendCommand [-iname <name>] <API query string>**
  Sends a query to the default or named interface. The query is specified as an ASCII string in API syntax.

  Example

```
NMSAPISendCommand "_cmd: get
_obj_type: network
_obj_id: networkID S compRoot
_scope: all
_attr_id: all
_filter:compID LEFT NI PM"
```

- **NMSAPIRecvReply [-iname <name>] [-out] [-var <array name>] [<timeout>]**

  Waits for and receives the next reply record from the server. A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts as a no-wait poll. If `-out` is specified, the received record is printed on standard output in an API-like format.

  If a variable name is specified with `-var`, the named variable is set as an associative array containing the received API fields. For API fields having an attribute name, the array key is set to:

  `<API label (without the :)>,<API attribute name>`

  and for the others, the key is set to:

  `<API label (without the :)>`

  If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

  Example

  Receving a GET command reply for a Network Model node could lead to the following values:

```
while NMSAPIRecvReply -var reply
do
    #...
    # ${reply[_obj_type]} is “node”
    # ${reply[_obj_id,compId]} is “PM R78”
    # ${reply[_attr,rawState]} is “INSV”
    #...and so on...
```

  It is possible to list all the array keys using Korn Shell’s `${!<variable name>[*]}` construct.

  The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred).

- **NMSAPISkipRestOfReply [-iname <name>]**

  This combination call waits for and ignores all further replies until the end of response. If it finds an error record, the `NMSEPI_RECORD_TYPE` environment variable is set to `ERROR`; otherwise it is set to `ENDRESP`.

- **NMSAPIGetRecord [-iname <name>] [-out] [-var <array name>]**
  Extracts the last received record's (`NMSAPIRecvReply`) record type (`NONE`, `REGISTER`, `RESPONSE`, `EVENT`, `ENDRESP`, `ERROR`, or `END`). It places this information in the `NMSEPI_RECORD_TYPE` environment variable (and prints it to standard output if `-out` is used) and resets the API field list to the beginning for `NMSAPIGetNextField`, `NMSAPIFindNextField`, and `NMSAPIFindNextAttr` (it can therefore be called several times to repeatedly scan the field/attribute list).

  If a variable name is specified with `-var`, the named variable is set as an associative array containing the received API fields. For API fields having an attribute name, the array key is set to:

  `<API label (without the :)>,<API attribute name>`

  and for the others, the key is set to:

  `<API label (without the :)>`

  If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

  It is possible to list all the array keys using Korn Shell's *`${!`*`<variable name>`*`[*]}`* construct.

  Example

```
while NMSAPIRecvReply
do
   NMSAPIGetRecord
   if [ "$NMSEPI_RECORD_TYPE" = "ENDRESP" ]
   then
      ... # got end of response
```

- **NMSAPIGetNextField [-iname <name>] [-out]**
  Extracts the next API field from the last received reply. The following environment variables are set:

  `NMSEPI_FIELD_LABEL`
  is the API field type (`_attr:`, `_end_resp:`, `_error:`, `_end:`, `_obj_type:`, `_obj_id:`, `_user_id:`, `_capability:`, `_inv_id:`, `_event_type:`, `_time:`, or `_sieve_id:`).

  `NMSEPI_FIELD_NAME`
  is the API field name element (`_ATTR:` and `_OBJ_ID:` only).

NMSEPI_FIELD_TYPE
is the API field type element (_ATTR and _OBJ_ID only).

NMSEPI_FIELD_VALUE
is the API field value.

Multiple-line attribute block values are returned as a single multiple-line value.

If -out is used, the field is printed in an API-like format on standard output. The return error code is in NMSEPI_RESULT (6 indicates there are no more fields).

Example

```
NMSAPIGetRecord
...
while NMSAPIGetNextField
do
   if [ "$NMSEPI_FIELD_LABEL" = "_attr:" \
       -a "$NMSEPI_FIELD_NAME" = "time" ]
   then
      echo "Time is: $NMSEPI_FIELD_VALUE"
...
```

- **NMSAPIFindNextField [-iname <name>] [-out] <label>**
  Locates and returns the next API line of the type specified by <label> (_attr:, _end_resp:, _error:, _end:, _obj_type:, _obj_id:, _user_id:, _capability:, _inv_id:, _event_type:, _time:, or _sieve_id:) from the API field list of the last received API reply. The environment variables are set as for NMSAPIFindNextField. If -out is used, the field is printed in an API-like format (but without the label) on standard output. The return error code is in NMSEPI_RESULT (6 indicates there are no more fields).The previous example can be rewritten as follows:

```
NMSAPIGetRecord
...
while NMSAPIFindNextField "_attr:"
do
   if [ "$NMSEPI_FIELD_NAME" = "time" ]
```

```
   then
      echo "Time is: $NMSEPI_FIELD_VALUE"
...
```

- **NMSAPIFindNextAttr [-iname <name>] [-out] <attr. name>**
  Locates and returns the next named attribute from the API field list of the last received API reply. The environment variables are set as for `NMSAPIFindNextField`. If `-out` is used, the field's value is printed on standard output. The return error code is in `NMSEPI_RESULT` (`6` indicates there are no more fields).The previous example can be rewritten as follows:

  Example
  `NMSAPIGetRecord`

```
...
if NMSAPIFindNextAttr "time"
then
   echo "Time is: $NMSEPI_FIELD_VALUE"
...
```

- **NMSAPIBindCallback [-iname <name>] <callback command>**
  Binds the specified DtKsh command string (typically a function invocation and its arguments) to the API interface. This command string will be executed whenever a new message is received from the server for this interface and the script is in an event loop (`NMSEPIEventLoop` or `XtMainLoop`). The following environment variables are available to the callback:

  `NMSEPI_INAME`
  is the API interface name of the callback.

  `NMSEPI_CB_REASON`
  is the reason for the callback (`LOST_CONNECTION`, `ERROR`, `EVENT`, or `RESPONSE`).

  `NMSEPI_RECORD_TYPE`
  is the returned record (API record type or `ALARM` if the received record is an Alarm, or `RAWSTATE` if it is a Raw State).

  `NMSEPI_ALARM_SEVERITY, NMSEPI_ALARM_EVENT, NMSEPI_ALARM_FAULT, NMSEPI_ALARM_TIME,`

NMSEPI_ALARM_COMPID, NMSEPI_ALARM_OPDATA
are the special alarm fields

NMSEPI_ALARM_TIME, NMSEPI_ALARM_COMPID, NMSEPI_RAW_STATE
are the alarm fields if a Raw State Change record is received

NMSEPI_API_SIEVEID
is the source Sieve ID for the received message, if any

In addition, the post-`Recv` functions (i.e., `NMSAPIGetRecord`, `NMSAPIGetNextField`) can be used in the context of this callback to extract other API fields and attributes from the received reply. The event loop is launched with `NMSEPIEventLoop` and never returns. As well, the DtKsh `XtInitialize` command must be called before using any DtKsh EPI initialization routines.

Example

```
function mycb {
   # ...Process the reply...
}
# ... launch commands and create sieves ...
NMSAPIBindCallback mycb
...
NMSEPIEventLoop # never returns
```

Specialized API access

This section describes additional commands as well as value-added versions of the Generic API Access commands that are built to interact with specific APIs and their features.

Alarm and Status API access

This area adds commands to interact with the Alarm and Status API in a more practical way. In addition to a number or combination calls (for example, to create an alarm sieve or to inject alarms), it also supports automatic extraction of the major alarm attributes to support most forms of network automation.

Alarm and Status API Access commands

The following Alarm and Status API Access commands are provided:

- **NMSGMDRAPIInit [-iname <name>]**
  This simplified form of the `NMSEPIInit` command initializes an API interface to the Alarm and Status API server (GMDR). If multiple API interfaces are to be used, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targetted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

- **NMSGMDRAPIConnect [-iname <name>] [-ssel | <host>]**
  Connects to the Service Selected `GMDR` server (if `-ssel` is specified), the specified location (`<host>`), or the local host (by default).

  Example

  ```
  NMSGMDRAPIInit -iname toto
  if NMSGMDRAPIConnect -iname toto -ssel
  then
     ... # we're connected
  ```

- **NMSGMDRAPICreateAlarmSieve [-iname <name>] [-all] [-attr <alarm event filters and attributes...>]**
  This combination call creates a simplified alarm sieve and waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the major alarm attributes (`compId`, `time`, `severity`, `event`, `faultCode`, and `operatorData`) of all of the received alarms. However, if `-all` is specified, all of the attributes are extracted. You can add event filter (`"_attr: eventFilter SS <attribute> <operator> <type> <value>"`) and specific attribute extraction (`"_attr: eventInfo S <attribute>"`) parameters in API syntax, using the `-attr` option and its values (note that each value may be multiple-lined).

Example

```
if NMSGMDRAPICreateAlarmSieve -attr \
   "_attr: eventFilter SS event EQ E SET" \
   "_attr: eventInfo S commentData"
then
   ... # sieve creation successful
```

The notifications from the sieve(s) can be collected with `NMSAPIRecvReply` or with `NMSGMDRAPIRecvAlarm` (with the `EVENT` record type).

Each Alarm & Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned as the `NMSEPI_API_SIEVEID` environment variable.

Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **NMSGMDRAPIRecvAlarm [-iname <name>] [-out] [-var <array name>] [<timeout>]**
  Enhances `NMSAPIRecvReply` by waiting for and receiving the next reply and automatically extracting the major alarm fields in their own environment variables (if the reply is the result of an alarm *get* or notification). The following environment variables are set:

  `NMSEPI_RECORD_TYPE:`
  is the API record type.

  `NMSEPI_ALARM_SEVERITY:`
  is the severity of the alarm.

  `NMSEPI_ALARM_EVENT`
  is the alarm event.

  `NMSEPI_ALARM_FAULT`
  is the fault code.

NMSEPI_ALARM_TIME
is the alarm time.

NMSEPI_ALARM_COMPID
is the component ID.

NMSEPI_ALARM_OPDATA
is the operator data.

For information on the legal values of these fields, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The other attributes, if any were specified, can be extracted with the `NMSAPIGetRecord`, `NMSAPIGetNextField`, `NMSAPIFindNextField`, and `NMSAPIFindNextAttr` commands. The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred).

Example

```
NMSGMDRAPICreateAlarmSieve -attr \
    "_attr: eventFilter SS event EQ E SET"
...
if NMSGMDRAPIRecvAlarm
then
   if [  "$NMSEPI_RECORD_TYPE"  = "EVENT" \
       -a "$NMSEPI_ALARM_SEVERITY" = "critical" ]
   then
      ... # handle the critical alarm
```

If `-out` is used, the default alarm fields are printed on the standard output stream as a sequence of at least six lines (for the `severity`, `event`, `faultCode`, `time`, `compId`, and `operatorData` respectively).

If a variable name is specified with `-var`, the named variable is set as an associative array containing the received API fields. For API fields having an attribute name, the array key is set to:

`<API label (without the :)>,<API attribute name>`

and for the others, the key is set to:

`<API label (without the :)>`

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

Note: Some lines may be empty if the corresponding field is not present. Also, `operatorData` can consist of more than one line.

- **NMSGMDRAPIFormatAlarm [-iname <name>] [-out] [terse|normal|full]**

  This command produces the last received alarm (from `NMSAPIRecvReply`, `NMSGMDRAPIRecvAlarm` or from a callback) in the Preside Multiservice Data Manager (MDM) Common Alarm Format (as used in the Alarm Display and Component Information Viewer tools). The alarm can be formatted in either `terse` (one line), `normal` (includes Operator Data) or `full` (all information) formats, the default is `full`. The `NMSEPI_ALARM_DISPLAY` environment variable is set to the resulting string.

  If `-out` is used, the resulting string is also printed on the standard output stream.

- **NMSGMDRAPIInjectAlarm [-iname <name>] <comp ID> <event> <severity> <fault code> <notification ID> <comment> [-time <time>] [-attr <other attributes...>]**

  Sends an Alarm Injection command with the following specified parameters:

  `comp ID`
  is the component ID.

  `event`
  is the alarm event (`set|clear|message`).

  `severity`
  is the severity of the alarm (`warning|minor|major|critical|cleared|indeterminate`).

  `fault code`
  is the fault code in `AAAACCCC` format.

  `notification ID`
  is the sequence number (if 0 or an empty string, a unique value will be

provided).

`comment`
is the comment data string.

As well, a time can be specified using `-time` (in *`yyyy mm dd hh mm ss`* format), and other attributes can be specified using `-attr` and its values (as one or more "`_attr: <name> <type> <value>`" per string). Unspecified attributes take default values (see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*).

Note: This is not a combination call since the command does not wait for the server's reply, which must then be explicitly received or ignored.

Example

```
NMSGMDRAPIInjectAlarm "MYRTR WEST3" "set" \
   major A0000001 23\
   "The router does not reply." \
   -attr "_block: _attr operatorData S
Try: 12, Timeout: 5, IP: 33.24.1.1
_end_block:"
NMSAPISkipRestOfReply
```

- **NMSGMDRAPICreateRawStateSieve [-iname <name>] [-attr <event filters...>]**

  This combination call creates a simplified raw state change sieve and waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the `compId`, `time`, and `rawState` attributes. It is possible to add an event filter ("`_attr: eventFilter SS <attribute> <operator> <type> <value>`") with the `-attr` option and its values (note that each value may be multiple-lined).

Example

```
if NMSGMDRAPICreateRawStateSieve -attr \
   "_attr: eventFilter SS rawState EQ E OOS"
then
   ... # sieve creation successful
```

The notifications from the sieve(s) can be collected with `NMSAPIRecvReply` or with `NMSGMDRAPIRecvRawState` (using the EVENT record type).

Each Alarm & Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned as the `NMSEPI_API_SIEVEID` environment variable.

Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **NMSGMDRAPIRecvRawState [-iname <name>] [-out] [-var <array name>] [<timeout>]**

  Enhances `NMSAPIRecvReply` by waiting for and receiving the next reply and automatically extracting the raw state change fields in their own environment variables (if the reply is the result of a node *get* or notification). The following environment variables are set:

  `NMSEPI_RECORD_TYPE`
  is the API record type.

  `NMSEPI_ALARM_TIME`
  is the notification time.

  `NMSEPI_ALARM_COMPID`
  is its component ID.

NMSEPI_RAW_STATE
is its raw state value.

For information on the legal values of these fields, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*.

Example

```
NMSGMDRAPICreateRawStateSieve -attr \
   "_attr: eventFilter SS rawState EQ E OOS"
...
if NMSGMDRAPIRecvRawState
then
   if [ "$NMSEPI_RECORD_TYPE" = "EVENT" ]
   then
      ... # handle the out-of-service component
```

If `-out` is used, the raw state fields are printed on the standard output stream as a sequence of three lines (for the `rawState`, `time`, and `compId` respectively). The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred).

If a variable name is specified with `-var`, the named variable is set as an associative array containing the received API fields. For API fields having an attribute name, the array key is set to:

```
<API label (without the :)>,<API attribute name>
```

and for the others, the key is set to:

```
<API label (without the :)>
```

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

Network Model API access

The Network Model (NM) API access allows you to initialize an API interface to the server and connect the interface to the host.

Network Model API Access commands

The Network Model (NM) API Access provides two additional commands:

- **NMSNMAPIInit [-iname <name>]**
  Initializes an API interface to the NM Server. If multiple API interfaces are to be used, specify a name in the command invocations for this

interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

- **NMSNMAPIConnect [-iname <name>] [-ssel | <host>]**
  Connects the interface to the Service Selected NM Server host (`-ssel`), or the named host, or the local host (by default).

  Example
  ```
  NMSNMAPIInit
  if NMSNMAPIConnect bcars561
  then
     ... # we're connected
  ```

Various Generic API Access commands can be used to send queries and receive their replies.

Host Group Directory Service API access

The Host Group Directory Service (HGDS) API Access allows you to extract data on the Passport Group configuration of Preside Multiservice Data Manager (MDM). Some value-added and combination calls are added to the Generic API Provider along with automatic extraction of the Passport Member information.

HGDS API Access commands

The following HGDS API Access commands are provided:

- **NMSHGDSAPIInit [-iname <name>]**
  Initializes an API interface to the Host Group Directory Server. If multiple API interfaces are to be used, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

- **NMSHGDSAPIConnect [-iname <name>] [-ssel|<host>]**
  Connects the interface to the Service Selected HGDS (`-ssel`), the named host, or the local host (by default).

  Example
  ```
  NMSHGDSAPIInit -iname hgds
  if NMSHGDSAPIConnect -iname hgds -ssel
  then
     ... # we're connected
  ```

- **NMSHGDSAPISendQuery [-iname <name>]**
  -group [<group name>]
  | -member [<member name>]
  | -child <group name>
  | -parent <member name>

  Sends an HGDS API query. Use `-group` to retrieve the named Passport Group (or all of them if no name is indicated), `-member` for the named Passport module (or all of them if no name is indicated), `-child` for the Passport host in the named group, or `-parent` for the groups containing the named host.

- **NMSHGDSAPIRecvReply [-iname <name>] [-out] [<timeout>]**
  Enhances `NMSAPIRecvReply` to automatically extract the Passport host information, if it is present. The following environment variables are set::

  `NMSEPI_RECORD_TYPE`
  is the reply record type.

  `NMSEPI_HGDS_NAME`
  is the Passport host or group name.

  `NMSEPI_HGDS_IPADDR`
  is the Passport host IP address (if applicable).

  The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred).

  Example:

```
if NMSHGDSAPISendQuery -iname hgds -child “$grp”
then
   while NMSHGDSAPIRecvReply -iname hgds
   do
      if ping $NMSEPI_HGDS_IPADDR
      then
         ... # Passport is reachable
```

Command access

Like the `cmccmd` utility, Command Access allows scripts to connect to Passport Groups and DPN-100 OAs (the command route), to send commands to the modules they contain, and to receive the replies. Command Access also provides several utility commands to assist with the parsing and identification of the command output. Command Access commands further communicate

with the Command Session servers (CMCFUN and CM) that correspond to the current `DISPLAY` environment variable (for example, for a script that uses the session servers in the current Preside Multiservice Data Manager (MDM) User Session and potentially uses its current group and OA connections in the command console).

For stand-alone (for example, CRON) or specific macros (for example, using Passport provisioning mode), it may be necessary to create a private Command Session for the execution of the script. Command Access provides the `NMSCmdSession` comand to start (and stop) a session. (For a description of `cmcwrap` and how to use it to write macros, see 241-6001-301 *Preside MDM Customization Administrator Guide*).

Command Access commands are used in the following sequence:

1 Initialize a command interface.

2 Connect to the Command Session server (CMCFUN).

3 Connect to one or more Passport Group(s) or OA(s), if necessary.

4 Send a command to a node in the connected groups and OAs.

5 Receive the command replies, either as a single string or one line at a time.

6 Use DtKsh or the provided commands to analyze the reply.

7 Disconnect from the command server.

8 Drop the command interface.

Multiple commands can be sent to the nodes, but individual command interfaces will handle these commands synchronously, one at a time. It is possible, however, to create multiple named command interfaces (each one resulting in its own connection to the server and therefore capable of independent, parallel commands).

Command Access commands

The following Command Access commands are provided:

- **NMSCmdSession [-display <DISPLAY>] start|stop**
  [-dpn <MDM host>] [-pp <MDM host>]
  [-idle <timeout>]

  Starts (`start`) or stops (`stop`) a private command session for the

specified <DISPLAY> name (must be a unique value workstation-wide for private sessions). This is equivalent to spawning the following command in the background:

```
/bin/env DISPLAY=<DISPLAY> \
   /opt/MagellanNMS/bin/loop
          -delay 3 \
               /opt/MagellanNMS/bin/icm \; \
               /opt/MagellanNMS/bin/cmcfun
```

Since loop is used, the session terminates automatically within 30 seconds if the script that has spawned it terminates without stopping it (be careful of middle shell processes). See 241-6001-301 *Preside MDM Customization Administrator Guide* for more information on the loop utility.

The calling script waits for two to three seconds after invoking this command (or performs whatever operations it wants to) before attempting to connect to the session to let the session servers initialize themselves properly.

If one of -dpn and/or -pp is specified, the matching service selection is applied to the new session so that the corresponding servers are used instead of the current workstation service selection. See NMSCmdSetServiceSelection to change the session's service selection.

If -idle is specified with a non-zero value, the time (in minutes) is passed to cmcfun which self-terminates automatically if the specified time elapses with no command activity. This ensures that device connection resources are freed if not used for an extended period of time. The next time the session is used, the required group connection needs to be re-created.

Example

```
# start a private command session and wait for it
# to initialize
NMSCmdSession -display mysession.$$ \
|| exit 1
```

```
sleep 3
# initialize an connect the command interface
# with -display mysession.$$ (see below)
```

- **NMSCmdInit [-iname <name>]**
  Initializes a command interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultCmd` to make it the default interface).

- **NMSCmdDrop -iname <name>**
  Drops the named command interface (frees allocated resources).

- **NMSCmdConnect [-iname <name>] [-display <DISPLAY>]**
  Connects the interface to the current Command Session servers (for interactive scripts used within a Preside Multiservice Data Manager (MDM) User Session or as an argument to `cmcwrap`) or the servers that correspond to the specified `DISPLAY` variable for a previously started alternate Command Session server set.

  Example
  ```
  NMSCmdInit
  if NMSCmdConnect -display mysession.$$
  then
     ... # we're connected
  ```

- **NMSCmdDisconnect [-iname <name>]**
  Disconnects the interface from the session servers. The interface may be reconnected to the servers from the same session or to servers from another session.

- **NMSCmdSetServiceSelection [-iname <name>]**
  [-dpn <MDM hostname>]
  [-pp <MDM hostname>]
  Controls the Service Selection settings for the Command Session that this interface is connected to. `-dpn` specifies the MDM host name for DPN Network Access. `-pp` specifies the MDM host name for Passport Network Access.
  This command is similar to using the Service Selection tool on the same session as the Session Servers. If other users (or programs/scripts) are

using that session, then they will also be impacted by this change. The Command Interface must be connected, see `NMSCmdConnect`) for this function to work.

-
- **NMSCmdSendConnect [-iname <name>] OA|GROUP <route> <name> <password>**

  Sends a connect request to the indicated route using the specified authentication information. If an error is found, the error text is the value of the `NMSEPI_OUTPUT_TEXT` environment variable.

  Note: Connecting to an already connected route results in an error, even though the route is available. Send a **""** command to the route to verify if a connection is already established. If the connection does not exist, the reply indicates an error.

  Example

  ```
  NMSCmdSendCommand myOA ""
  NMSCmdSkipRestOfReply \
  || NMSCmdSendConnect OA myOA myCap aq1sw2
  if [ $NMSEPI_RESULT = 0 ]
  then
     ... # we're connected to the route
  ```

  Note: If the script is to be invoked from the Preside Multiservice Data Manager (MDM) Command Console, you can use `"$CMC_CURRENT_ROUTE"` (with the quotes as indicated) here and in `NMSCmdSendCommand` to indicate the current Command Console route.

- **NMSCmdSendDisconnect [-iname <name>] <route>**
  Disconnects the interface from the named route.
- **NMSCmdSendCommand [-iname <name>] <route> <command>**
  Sends a command to a node through the specified connected route. The name of the destination node is typically the first token of the command.

  Example

  ```
  if NMSCmdSendCommand myOA "R78 d"
  then
     ... # command sent
  ```

Note: The special command routes of the Command Console ($ for macro access, @ for the SNMP Command Framework, and * for the Passport wild-card group) can all be used as routes from EPI though, obviously, there is no need to first connect to them.

Note: Be careful when specifying, in the node command, special characters (for example, `*` or `?`) which may be interpreted and substituted by DtKsh. You should quote the command string as a whole.

- **NMSCmdRecvFullReply [-iname <name>] [-out]**
  Waits for and receives the complete reply to the previous node command. The replied text is available in the `NMSEPI_OUTPUT_TEXT` environment variable (and printed to the standard output if `-out` is used).

  Example
  ```
  NMSCmdSendCommand myOA "R78 d"
  if NMSCmdRecvFullReply
  then
     echo "$NMSEPI_OUTPUT_TEXT" | grep "pe"
  ...
  ```

- **NMSCmdRecvNextLine [-iname <name>] [-out] [<timeout>]**
  Waits for and receives the next line of the last reply of the issued node command. A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts as a no-wait poll. The replied text is available in the `NMSEPI_OUTPUT_TEXT` environment variable (and is printed to the standard output if `-out` is used). The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred; `6` indicates there are no more lines). The previous example can be rewritten as follows:

  Example
  ```
  NMSCmdSendCommand myOA "r78 d"
  while NMSCmdRecvNextLine
  do
     if [[ "$NMSEPI_OUTPUT_TEXT" = *pe* ]]
     then
        echo "$NMSEPI_OUTPUT_TEXT"
  ...
  ```

- **NMSCmdRecvNextChunk [-iname <name>] [-out] [<timeout>]**
  Waits for and receives the next chunk of the last reply of the issued node command. A chunk is most efficiently processed by EPI and may contain

multiple lines of text (it can even end in the middle of a line). A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts as a no-wait poll. The replied text is available in the `NMSEPI_OUTPUT_TEXT` environment variable (and is printed to the standard output if `-out` is used). The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred; `6` indicates there are no more lines). The previous example can be rewritten as follows:

Example

```
NMSCmdSendCommand myOA "r78 d"
while NMSCmdRecvNextChunk
do
   if [[ "$NMSEPI_OUTPUT_TEXT" = *pe* ]]
   then
      echo "$NMSEPI_OUTPUT_TEXT"
...
```

- **NMSCmdSkipRestOfReply [-iname <name>]**
  This combination call waits for and ignores all replies until the end of response.

- **NMSCmdGetNumColumns [-iname <name>]**
  Returns the number of blank (space or tab) separated columns in the previously received reply line (`NMSCmdRecvNextLine`). The column count is returned by the routine instead of an error code as well as in the `NMSEPI_NUM_COLUMNS` environment variable.

- **NMSCmdGetColumn [<-iname <name>] [-out] <column> [...] [<target>]**

  Without `<target>`, returns the indicated blank (space or tab) separated column of the previously received reply line (`NMSCmdRecvNextLine`) as the `NMSEPI_OUTPUT_COLUMN` environment variable (and is printed to the standard output if `-out` is used). If '`...`' is specified, all of the remaining columns starting at the given column are returned (blank separated). Column indexes start at 1.

  Example

  ```
  NMSCmdSendCommand myPassp "pp1 d fruni/101"
  while NMSCmdRecvNextLine
  do
     NMSCmdGetColumn 1
     att="$NMSEPI_OUTPUT_COLUMN"
  ```

```
   NMSCmdGetColumn 3 ...
   val="$NMSEPI_OUTPUT_COLUMN"
   if [ "$att" = "operationalState" ]
   then
      echo "State: $val"
...
```

If <target> is specified, this command also checks whether the column has the same contents as the target and returns accordingly.

Example

```
NMSCmdSendCommand myPassp "pp1 d fruni/101"
while NMSCmdRecvNextLine
do
   if NMSCmdGetColumn 1 "operationalState"
   then
      NMSCmdGetColumn 3 ...
      echo "State: $NMSEPI_OUTPUT_COLUMN";
...
```

- **NMSCmdPatternMatch [-iname <name>] [-out] [-g] <pattern> [<substitute>]**

  This is the functionality of NMSEPIPatternMatch applied to the last received command response (full, line, or chunk).

  Example

  ```
  NMSCmdSendCommand myOA "dir"
  while NMSCmdRecvNextLine
  do
     if NMSCmdPatternMatch ".*pe.*"
     then
        echo "$NMSEPI_OUTPUT_TEXT"
  ...
  ```

- **NMSCmdSendDestRequest [-iname <name>][OA|GROUP|ALL]**

  This special utility command requests a list of all OA or Passport Group (or both) types of available routes. It corresponds to the cmccmd list command.

- **NMSCmdRecvNextDest [-iname <name>] [-out] [<timeout>]**

  Waits for and receives the next reply to a NMSCmdSendDestRequest command. The reply is returned with the following environment variables:

NMSEPI_OUTPUT_TEXT
is the full reply text line.

NMSEPI_DEST_NAME
is the route name.

NMSEPI_DEST_TYPE
is the route type (OA or GROUP)

NMSEPI_DEST_STATE
is the route connection state (CONN, AUTH, or -).

The return error code is in NMSEPI_RESULT (5 indicates a timeout occurred; 6 indicates the end of response).

Example

```
NMSCmdSendDestRequest GROUP
while NMSCmdRecvNextDest
do
   NMSCmdGetColumn 2
   if [ "$NMSEPI_OUTPUT_COLUMN" = "CONN" ]
   then
      ... # route is available
```

- **NMSCmdBindCallback [-iname <name>] [-chunk|-ppcomp] <callback command>**

  Binds the specified DtKsh command string (typically a function invocation with its arguments) to the command interface. This command string is executed whenever a new line (default) of reply, a new chunk (-chunk) of reply, or a new Passport component (-ppcomp) reply is received from the server for this interface. The following environment variables are available to the callback:

  NMSEPI_INAME
  is the callback command interface name.

  NMSEPI_CB_REASON
  is the reason for the callback (LOST_CONNECTION, ERROR, ENDRESP, or RESPONSE or FLOW_CALLBACK)

NMSEPI_OUTPUT_TEXT
is the next line (default) or chunk (`-chunk` mode) of output from the command.

NMSEPI_PPCOMPID
is the the component ID of the next Passport component (`-ppcomp` mode).

The text manipulation commands previously described (`NMSCmdGetNumColumns`, `NMSCmdGetColumn`, and `NMSCmdPatternMatch`) are also available in the context of the callback in line and chunk mode. In Passport component mode, use `NMSCmdGetPPCompID, NMSCmdResetPPCompAttrs, NMSCmdGetFirstPPCompAttr, NMSCmdGetNextPPCompAttr,` and `NMSCMDFindNextPPCompAttr` to extract the Passport component information. The event loop is launched with `NMSEPIEventLoop` and never returns. As well, the DtKsh `XtInitialize` must be called before using any DtKsh EPI initialization routines. The return error code is in `NMSEPI_RESULT` (6 indicates the end of response).

Example

```
function mycb {
   # ...Process the reply...
}
...
NMSCmdSendCommand myOA “r72 q serv”
NMSCmdBindCallback mycb
...
NMSEPIEventLoop # never returns
```

- **NMSCmdRecvNextPPComp [-iname <name>] [-out] [-var <var name>] [<timeout>]**

  `NMSCmdRecvNextPPComp` is a form of `NMSCmdRecvFullReply` that waits for and receives the next Passport component reply to the previous command (i.e. the result of a `list` or `display` command). The name of the received component is available in the `NMSEPI_PPCOMPID` environment variable (and printed to the standard output if `-out` is used) and any error message is available in the `NMSEPI_OUTPUT_TEXT` environment variable.

Note: Make sure you always specify the `-notab` (no tabular output) option when sending a Passport CAS display command with wildcards if this command is to be used to extract the replies.

The `NMSCmdGetPPCompID`, `NMSCmdGetNextPPCompAttr`, and `NMSCmdFindNextPPCompAttr` commands can then be used to extract the replied component information. In addition, if a variable is specified with the `-var` option, the named variable is set as an associative array whose elements are the individual component attribute values. The element key is the returned attribute name. If the attibute is a list, vector or array, the first index is added to the key with a coma as separator (for example, `${rep[pktFromIfByPrio,ep0]}`). For two dimensional arrays, the entry is created with the attribute name as key and the column title as value. Another entry is created with "`<attribute name>,<row title>`" as key and the list of column labels as value. The remaining entries for this attribute have "`<attribute name>,<row label>`" as index and the list of corresponding columns as values. Finally, the entry key can also be one of the following special values; `Message`, contains any message emitted by Passport not part of of an attribute value (i.e. error messages), `CompID`, is the returned component's name.

It is possible to list all the array keys using Korn Shell's `${!<variable name>[*]}` construct.

Example

```
NMSCmdSendCommand myGroup \
                "TOTO display shelf card/* utilization"
while NMSCmdRecvNextPPComp -var reply
do
   echo "Card: $NMSEPI_PPCOMPID"
   echo "Average CPU: ${reply[cpuUtilAvg]}"
...
```

Other examples, using a display of the `Module-Vcs` and Passport `Shelf-Card` components, are:

```
# Simple attribute:
   ${reply[cpuUtil]} ... "5 %"
# SET value
   ${reply[highPriorityPacketSizes]} ..
                "16 32 64 128 256..."
# Vector values
```

```
    ${reply[memoryUsage,fastRam]} = "0 kbyte"
    #... and so on ...
# 2-D array values
    ${reply[windowSize]} ... "throughputClass"
    ${reply[windowSize,packetSize]} ...
              "0 1 2 3 4 ..."
    ${reply[windowSize,16]} ... "4 4 4 4 4 ..."
    #... and so on ...
# error
    ${reply[Message]} ... "   Component is disabled."
    ${reply[localMsgBlockCapacity]} ... "? kbyte"
```

- **NMSCmdGetPPCompID [-iname <name>] [-out] [-var <var name>]**
  `NMSCmdGetPPCompID` extracts the name of the last received Passport component in the `NMSEPI_PPCOMPID` environment variable. If `-out` is specified, the name is also output on the standard output stream . If an associative array variable name is specified with `-var`, the array is also filled with the component's attributes as described for the `NMSCmdRecvNextPPComp` command. Finally, the list of Passport component attribute is reset to the beginning for the `NMSCmdGetNextPPCompAttr` and `NMSCmdFindNextPPCompAttr` commands.

  Example

```
...
while NMSCmdRecvNextPPComp -iname alt -var reply
do
    NMSCmdGetPPCompID -iname alt
    echo "Component: $NMSEPI_PPCOMPID"
...
```

- **NMSCmdGetNextPPCompAttr [-iname <name>] [-out]**
  `NMSCmdGetNextPPCompAttr` extracts the next attribute from the last received Passport component. The attribute name (and index, see the discussion on the associative array in `NMSCmdRecvNextPPComp`) is returned as the `NMSEPI_PPATTR_NAME` and its value as the `NMSEPI_PPATTR_VALUE` environment variables. If `-out` is specified, both values are also output on the standard output stream with a space as separator.

Example

```
...
while NMSCmdGetNextPPCompAttr
do
    echo "Attribute: $NMSEPI_PPATTR_NAME"
    echo "Value: $NMSEPI_PPATTR_VALUE"
...
```

- **NMSCmdFindNextPPCompAttr -iname <name>] [-out] <attribute name>**

  `NMSCmdFindNextPPCompAttr` extracts the next attribute from the last received Passport component whose name (in index, see the discussion on the associative array in `NMSCmdRecvNextPPComp`) matches the specified value. The attribute name/index is returned as the `NMSEPI_PPATTR_NAME` and its value as the `NMSEPI_PPATTR_VALUE` environment variables. If `-out` is specified, both values are also output on the standard output stream with a space as separator.

  Example

```
...
if NMSCmdFindNextPPCompAttr "bytesSent"
then
    NMSEPILongArith $prev - $NMSEPI_PPATTR_VALUE
    echo "Delta: $NMSEPI_RESULT"
    prev=$NMSEPI_PPATTR_VALUE
...
```

- **NMSCmdDoCommandFile [-iname <name>] [-out] [-trace[:[C][R][E][X]]] [-test] [-cb] [-bestEffort] [-avl <var name>] [<var>=<value> ...] <dest> <file path>**

  NMSCmdDoCommandFlow [-iname <name>] [-out] [-trace[:[C][R][E][X]]] [-test] [-cb] [-bestEffort] [-avl <var name>] [<var>=<value> ...] <dest> <command flow>

  These calls support the synchronous execution (commands and replies) of command flows from the file identified by `<filePath>` or the string identified by `<commandFlow>`. A command flow is a sequence of device commands (one per line) to be executed as one. The file may be specified as a formal path (starting with '/', '.' or '. .', or '~' which is

automatically substituted to the user's `HOME` account), or as a relative path in which case the file is automatically searched for; first, in `$HOME/MagellanNMS/<file path>`, then `/opt/MagellanNMS/cfg/<file path>`, and finally `/opt/MagellanNMS/lib/<file path>`.

All commands are executed through the named destination (if empty -- ""--, the destination must be prefixed to every device command in the flow). The execution of the flow terminates at the first failure unless `-bestEffort` is specified. Variable substitution on each command in the flow is supported with the associative array named with `-avl` as the source of the variable name-value pairs. Additional (or overriding) variable assignments can be specified with `<var>=<value>` arguments.

The flow may produce some output text and numerical results available as the `NMSEPI_OUTPUT_TEXT` and `NMSEPI_RESULT` environment variables respectively. The output text is produced from the flow using the `@PRINT` directive (see below). If `-trace` is specified, the executed device commands are also added to the output text. This output text contains a `#*` prefix for plain commands and `#?` for conditional commands followed by their output and a `#? END`, `#? FAILED`, `#* END`, or `#* FAILED` to indicate the end of of the corresponding command. The output result is produced from the flow by the `@EXIT` directive or the

NMSCmdTerminateCommandFlow command in callback mode (see below). You can control specific tracing by using the following optional modifiers:

C traces only the executed plain commands. Control construct commands (@SWITCH, @FOREACHPP, ...) are not traced.

R traces only the plain command responses (also excludes control constructs).

E traces only error responses from plain commands (also excludes control construct commands).

X traces control construct commands and responses.

You can specify one or more modifier letters following a colon. You can also control tracing from the flow itself by using the @TRACE construct.

The output processing command (NMSCmdGetNumColumns, NMSCmdGetColumn, and NMSCmdPatternMatch) can also be used to examine the output.

If -cb is specified, the output text is not returned in the NMSEPI_OUTPUT_TEXT variable. Instead, the callback bound to the command interface by NMSCmdBindCallback is invoked for each line of output (including the @PRINT and #? and #* comments) where it is available in the usual manner (with reason RESPONSE). From the callback, it is possible to force the termination of the flow execution by invoking the NMSCmdTerminateCommandFlow command with the desired numerical result (as if @EXIT had been reached in the file).

Note that in callback mode, even though the output is returned line by line through the bound callback, the execution of the flow is still synchronous and no other EPI actions will be performed until the flow execution has completed. In other words, the flow/file execution function will not return until the flow execution is complete during which the bound callback will be invoked for each output line. Once the flow

execution completes, the bound callback is called with reason `ENDRESP` on success or ERROR on failures, unless the execution was explicitly terminated with `NMSCmdTerminateCommandFlow`.

If `-test` is specified, the command flow/file is executed in test mode as described later.

Command flows consist of a list of device commands with, optionally, substitution variables identified by a dollar sign '`$`' (to specify a plain `$`, escape it as '`\$`'). For example, the following Passport command sets the `committedInformationRate` of a FrameRelay DLCI (read as one line):

```
$name set FrUni/$fruni Dlci/$dlci Sp\
            committedInformationRate $cir
```

A flow containing such a line should be executed with an attribute-value associative array containing at least:

```
typeset -A avl
avl[name]=NODER16
```

```
avl[fruni]=120
avl[dlci]=25
avl[cir]=56000
```

Other forms of substitution variable specification include;

${<variable name>**}**	same as without the brackets.
$!**<variable name>, or **${!<variable name>**}**	(strict substitution) When used in device commands (does not apply to the special directives below), the command will be skipped (silently not executed) if the specified variable has no associated value or is empty. Note: This form is only available in actual device commands.
${<variable name>**:-** <default value>**}**	If the specified variable has no associated value or is empty, substitutes the specified default value instead.
$%	This special internal variable holds the contents of the last executed `@SWITCH` command or the value of the matched `\(\)` subpattern of the last executed `@case` command (see below)
$%%	This special internal variable holds the pattern-matched contents (whole or sub-pattern) of the last executed `@IF` command (see below).

$?	This variable contains the numerical result of the last issued macro (@DO) or flow (@INCLUDE/@RUN).
${<variable name>**[**<index>**]}**	This represents an associative array value in the Flow language itself (not to be mistaken by array values in the scripting language). Such values can be directly set (@SET, @DEFINE, @LOCAL) or provided by specialized commands (@FOREACHPP, @SPLITCOMP, @SPLIT). When used, both the variable name and the index can also be substituted variables (for example, ${array[$i]} in a loop that increments the value of $i). The ! and :- constructs also apply to the array entry (for example, ${array[$i]:-0} defining 0 as the default value for the entry).

Command flows also support special processing directives as indicated in the following table:

@PRINT <string>	appends the specified text (after variable substitution) to the command output (NMSEPI_OUTPUT_TEXT).
@FORMAT <multi-line string...> **@END**	same as @PRINT but for a multi-line piece of text.
@EXIT [<code>]	terminates the flow's execution with the specified result code (NMSEPI_RESULT). If no @EXIT directive is executed, the flow will have its result set to 0 on success or 4 if the flow was terminated by a failed command.

@RETURN [<code>]	like @EXIT but when invoked from an included file (see @INCLUDE), only terminates the execution of the included file and returns to the calling flow/file. If used from within a macro or included/run flow, the return value is available as the $? variable from the calling flow.
@TRY [<command>]	executes the command and ignores its possible failure, even if -bestEffort was not specified. If the command is omitted, sets the current operating mode for subsequent commands as if -bestEffort had been specified. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.
@CRITICAL [<command>] or **@CRIT** [<command>]	executes the command and terminates the flow if the command fails, even if -bestEffort was specified. If the command is omitted, sets the current operating mode to critical for subsequent commands. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.
@TRACE [<command>]	traces this command, even if -trace is not specified. If the command is omitted, sets the current operating mode to trace (as if -trace had been specified) for subsequent commands. Other modifiers (@TRY/@CRITICAL, @LOG/@NOLOG) may also be specified.

@NOTRACE [<command>]	does not trace this command, even if -trace is specified. If the command is omitted, sets the current operating mode to no-trace for the subsequent commands. Other modifiers (@TRY/@CRITICAL, @LOG/@NOLOG) may also be specified.
@LOG [<command>]	logs this command as long as a log file has been defined (NMSCmdOpenCommandLog or @LOGFILE). If the command is omitted, sets the current operating mode to logging. Other modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE) may also be specified.
@NOLOG [<command>]	does not log this command, even if logging was enabled. If the command is omitted, sets the current operating mode to no-logging for the subsequent commands. Other modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE) may also be specified.
@LOGFILE [[**+**]<log file path>]]	controls command logging (see NMSCmdOpenCommandLog). Without arguments, this is equivalent to @LOG. If the log file is identified, the issued commands are logged to it from this point on (unless modified by @NOLOG). If the file path is prefixed with the plus (+) sign, the logs are appended to the file if it exists (else the file is overwritten with the new logs).

```
@IF <var.> [==|!=
            <patterns>]
or
@IF <var.> <|>|<=|>=
            <value
<commands>
...
[@ELSE
<commands>
...]
@END
```

evaluates the specified variable expressions and executes the first command block if it finds that it matches one (==) or does not match any (!=) of the patterns, first form, or compares (numeric or string, as appropriate), the second form, to the specified value. Otherwise, the command block following the @ELSE directive is executed, if any. If just the variable expression is specified, the test is positive if the expanded value is not empty. If the test was a pattern matching one, the value of the sub-pattern is available as the $%% internal variable.

```
@FOR <var> <from> <to>
       [<increment>]
or
@FOR <var> IN
       <token list>
<commands>
...
@END
or
@FOR <var> KEYS
       <array name>
<commands>
...
@END
```

executes the command block repeatedly, in the first form, while incrementing the named (<var>) numerical variable in the AVL from <from> to <to> in jumps of <increment> (defaults to 1), or in the second form, iterating the variable across the list of blank separated tokens. The variable ($<var>) can be used in the command block. The loop can be terminated prematurely by calling the @BREAK command from within the command block. Loops can be nested. The third form allows the variable to scan the existing indicies of the named associative array (for example, the Passport attribute values from @FOREACHPP).

```
@FOREACHPP <var>
        <Passport
         list/display
         command>
<commands>
...
@END
```

executes the specified Passport list command, and then iterates over the returned component names, assigning its name to the named variable and executing the command block. The loop can be terminated prematurely by calling the `@BREAK` command from within the command block. Loops can be nested. If the Passport command was a display one, the extracted attribute values are also available as the entries of an associative array of the same name as the specified variable. They are provided in the same way as from the `NMSCmdRecvNextPPComp` function. If wild-cards are used, make sure you specify the -notab display command option.

```
@FOREACHLN <var>
        <command>
<commands>
...
@END
```

executes the specified command, and then iterates over the returned output line by line, assigning each one to the named variable and executing the command block. The loop can be terminated prematurely by calling the `@BREAK` command from within the command block. Loops can be nested.

```
@BREAK
```

invoked from within a `@FOR/ @FOREACHPP/@FOREACHLN/ @WHILE/@WHILDO` loop construct, it terminates the enclosing loop prematurely. In other situations, it acts like `@RETURN` with no return code.

@CONTINUE	invoked from within a @FOR/ @FOREACHPP/@FOREACHLN/ @WHILE/@WHILDO loop construct. This command causes the loop to immediately iterate to the next cycle. Like @BREAK, the following loop-code is not executed but unlike @BREAK, the loop is not abandoned.
@CB <text>	if the Flow is running in callback mode (-cb option), the callback is invoked with the specified text as the output text (NMSEPI_OUTPUT_TEXT). The callback reason (NMSEPI_CB_REASON) is then set to FLOW_CALLBACK. The callback can interpret this text as convened and, in reply, set or reset AVL variables with NMSCmdSetFlowCBAVL before returning so the flow can use the results. This may be used to query another system or a user for a value needed in the flow processing (Wizzard).

```
@SWITCH <test command>
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

executes the test command and executes the commands following the first @CASE whose patterns match the output of the test command (the optional commands that follow the @SWITCH before the first @CASE are always executed). Only one @CASE block in the @SWITCH construct is executed. @SWITCH blocks can be nested. Patterns are specified in GREP format with a '|' separating alternatives. If no pattern is specified, the @CASE block accepts any output. The @SWITCH command is traced to ouput, if enabled, as:

```
#? <test command>
<command output...>
#? END
```

(If the command could not be executed, the last line will be **#? FAILED** instead).

The @CASE patterns may contain one subexpression (\(\)), each of whose matched value is available in the following code as the $% substitution variable. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @SWITCH.

```
@SWITCHVAL <value expr.>
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

This construct behaves much like @SWITCH but instead of getting its pattern match target value from a device command output, that value is directly specified as a parameter.

```
@WHILE <var.>
       [==|!=|<|>|<=|>=
       <patterns/
value>]
<commands>
...
@END
```

repeatedly executes the command block as long as the text (same as `@IF`) succeeds. The loop can be broken prematurely by invoking `@BREAK`.

```
@INCLUDE <file path>
or
@RUN <file path>
```

The named file is included and processed as though its contents were part of the current flow.Variable definitions (`@DEFINE`) performed in the file invoked by `@INCLUDE` apply to the calling flow. Variable definitions performed in the file invoked by `@RUN` are ignored upon return. If the file is identified as a relative path, the standard Preside Multiservice Data Manager (MDM) search path applies (see above). If the file cannot be read, the flow's execution terminates unless `-bestEffort` was specified or the `@TRY` prefix is used. The usual modifiers (`@TRY`/`@CRITICAL`, `@TRACE`/`@NOTRACE`, `@LOG`/`@NOLOG`) can be specified before `@INCLUDE`/`@RUN`.

@MACRO <name> [<parameters...>] <code text> ... **@ENDMACRO**	defines a new macro that can be executed later with @DO/@IFDO/@WHILEDO. The macro can be given a number of positional argument names to be provided when executed (the last specified parameter name gets all the remaining arguments passed). These parameter names have local scope to the macro (as if defined with @LOCAL). The macro can be recursive. Just like @INCLUDE/@RUN, it can be terminated by @RETURN which specifies a return code available as the $? variable to the caller. Variables defined/set by the macro have the same scope as the caller unless defined with @LOCAL. Macros must be defined before they are used. The macros themselves have global scope and can be redefined.
@DO <macro name> [<arguments...>]	invokes the named defined macro. The specified arguments are assigned (local scope to the macro) to the macro's parameter names -- the name gets the left over arguments). The @RETURNed value from the macro is available as the $? internal variable. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified before @DO.

```
@IFDO <macro name>
      [<arguments...>]
<command>
...
[@ELSE
<commands>
...]
@END
```

merges the functionality of the @IF and @DO constructs. Executes the macro as for @DO then performs either the first command block if the macro returns a 0 result, else performs the second (@ELSE) command block if any. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @IFDO.

```
@WHILEDO <macro name>
      [<arguments...>]
<commands>
...
@END
```

merges the functionality of the @WHILE and @DO constructs. Repeatedly executes the macro as for @DO then the command block as long as the macro returns a 0 result. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @WHILEDO. As with @WHILE, the loop can be broken with @BREAK invoked in the command block.

```
@DEFINE <name> <value>
```

defines or redefines a variable name. The new value is available in the current command block and the blocks it invokes, notably for included files. For example, if a @DEFINE is invoked in a @CASE block, the modified value applies to the commands in that block but not in the commands that follow the @SWITCH construct for that @CASE. Similarly, @DEFINEs used in included files have no effect on the calling command block.

@UNDEFINE <name>	Contrary to @DEFINE, @SET, and @LOCAL, undefines the named variable. All matching associative array values are also undefined. To undefine a single entry in the array, specify its full name (for example, @UNDEFINE <array>[<index>]).
@LOCAL <name> [<value>]	Like @DEFINE but the new variable has local scope (it will not replace nor exist in the caller's scope, for example when used from within a macro or included flow). The local scope also applies to associative arrays by that name. This is useful when defining macros that need variables for which you do not want to override existing values.
@SET <name> <value1> [+|-|*|/|%|~ <value2>]	like @DEFINE but sets the variable to the result of the numerical expression (+, -, *, /, % -- remainder). The ~ operator performs a pattern match using the second value as a GREP style pattern pattern list (| separated). The variable is set to the matching portion or to nothing. If the pattern contains a \(\) delineated sub-pattern, it is that matching sub-pattern that is used as the new value. If only the name and first value are specified, the effect is the same as @DEFINE.

@SPLIT[**(**<separators>**)**]
<array name>
<string>

This tokenizes the specified string. The individual tokens are assigned to indexed elements of the named associative array (starting at `1`). The actual variable's value is the number of resulting tokens. By default, tokenization is done on blanks. Alternatively, the separator characters can be provided between brackets. For example, the following will print the individual applications in a Passport AVL, one per line:

```
@SPLIT(, \t\n) app $avl
@FOR i 1 $app
     @PRINT ${app[$i]}
@END
```

@SPLITCOMP <array name> <component ID>

analyzes the specified component ID and provides the results in the named associative array. The following examples are based on the component EM/TOTO LP/2 Ds1/0.

- **$**<array> the component ID in API format (EM TOTO LP 2 DS1 0)
- **${**<array>**[_MOD]}** the module name (TOTO)
- **${**<array>**[_SUB]}** the subcomponent portion (minus first level) (LP/2 DS1/0)
- **${**<array>**[_PAR]}** the parent subcomp portion (minus last level) (EM/TOTO LP/2)
- **${**<array>**[**<category>**]}** the relative instance value for that level (EM -> TOTO, LP -> 2, DS1 -> 0)

@WAIT <nb seconds>

blocking wait for the specified number of seconds.

```
@ASK <name>
     [:E|:I|:S]
     [/validation
patterns/]
     [=<default value>]
     <prompt string>
@CASK <name> :
     [:E|:I|:S]
     [/validation
patterns/]
     [=<default value>]
     <prompt string>
```

asks you (standard input) for the value of the named variable using the specified string as a prompt. The expected type of the value can be specified as one of the following for a plain string:

`:E` for a string token enumeration
`:I` for an integer
`:S` (the default)

You can use a pattern between two forward slashes (//) or vertical bars (`|`). Specify the pattern so that EPI validates the entered value and prompts if there is no match. For enumeration, the pattern is a blank/coma separated list of words to match (for example, `/on off/`). For integers, the pattern is a blank/coma separated list of numeric values or ranges (for example, `/1,3,5,10-15,20/`). For strings (default) the pattern is an extended GREP style pattern list with `|` between alternative patterns (for example, `/.* Ds1\/.*|.* E1\/.*/` -- The forward `/` in the pattern is escaped with a single backwards `\` so it is not included as the end of the pattern). If you specify a default value, this value is set to the variable if you enter nothing (carriage return only). If you do not specify a default value, you are prompted when you enter an empty string.

`@CASK` (conditional ask) is similar to `@ASK` except that it does not prompt if the variable already has a non-empty value.

`# <comment>`	comment line.
`<device command>`	actual device command, optionally with embedded variables (`$` prefix) invoked after substitution. If the command fails, the flow execution terminates (no `-bestEffort` nor `@TRY` prefix). The command is traced to ouput, if enabled, as: **`#*`** `<device command>` `<command output...>` **`#* END`** (If the command indicated an error, the last line will be **`#? FAILED`** instead)

Note: Note that `@TRY`, `@CRITICAL`, `@TRACE`, and `@NOTRACE` can be combined. They can also be used with the `@INCLUDE` directive. Also, the command specified with `@SWITCH` can also start with `@TRACE` or `@NOTRACE`.

Example

Assuming a file (examp.tmpl) containing:

```
# Frame Relay QOS example
@SWITCH $name l FrUni/$fruni Dlci/$dlci
@CASE failed|ERROR
  @PRINT $name FrUni/$fruni Dlci/$dlci does not exist!
  @EXIT 1
@CASE
  $name set FrUni/$fruni Dlci/$dlci Sp cir $cir
  $name set FrUni/$fruni Dlci/$dlci Sp bc $bc
  $name set FrUni/$fruni Dlci/$dlci Sp be $be
  @IF $lmi
    $name set FrUni/$fruni Lmi procedures $lmi
    $name set FrUni/$fruni Lmi side $lmiside
  @END
@END
```

This flow can be executed with:

```
if ! NMSCmdDoCommandFile -avl avl "*" examp.tmpl
then
    print "Failed!!!\n$NMSEPI_OUTPUT_TEXT"
    exit 1
fi
```

If, instead, the contents of the file above are stored/built in a shell variable (for example, FLOW_STRING), the flow can then be executed with the following (note the quotes around the $FLOW_STRING specification to avoid the shell from absorbing the carriage returns between commands):

```
if ! NMSCmdDoCommandFlow -avl avl "*" "$FLOW_STRING"
then
    print "Failed!!!\n$NMSEPI_OUTPUT_TEXT"
    exit 1
fi
```

Example

The following are small examples of flow code usage:

```
# determine the Passport version and save it
# for later use
@SWITCH $name d software avl
@CASE base_\([^ ,]*\)
  # $% contains the last \(\) match (the base version)
  @PRINT Passport version : $%
  @DEFINE ppversion $%
  # set variables accordingly (Tm subcomponent
  # introduced?)
  @IF $ppversion == CA.*|CB.*
    @DEFINE tm Tm
  @ELSE
    @DEFINE tm
  @END
@END
...
# use the variable defined above and set the
# p1 parameter to a default value if not set
$name set AtmIf/$atm Vcc/$vpci Vcd $tm txTdp 1 \
              ${p1:-64000}
...
```

```
# create multiple DS1 channels with @FOR
@FOR chan 0 24
  $name add Lp/$lp DS1/$ds1 Chan/$chan
  $name Lp/$lp DS1/$ds1 Chan/$chan timeslots $chan
  # run a secondary flow to create a FrUni
  # for each channel (the flow has access to
  # the current AVL including the $chan variable)
  @RUN addFrUni
@END
```

CAUTION

DoEPITemplate

The DoEPITemplate helps you use and invoke command flows by handling scripting aspects. You only need to create the required flow text. The utility handles everything else, including the Passport configuration pre and post-amble for the configuration flows.

See “DoEPITemplate Utility” (page 607) for the description of the DoEPITemplate utility.

Test Mode

To test command files/flows without executing them, for example,while developing complex configuration templates, you can invoke the command with the `-test` option. This allows you to test the variable substitution, the `@SWITCH`/`@CASE` pattern matching and the general execution flow of the commands with or without actually executing them. In test mode, commands to be excuted are traced to the standard error stream (after variable substitution and prefixed by its line number) then a prompt usually asks for confirmation if it should be executed. The prompt depends on the command being executed:

@FOR <variable name> <from> <to> [<increment>]
or
@FOR <variable name> **IN** <token list>
on each iteration, the prompt offers to execute the command block or not::

```
> Confirm command block execution? ([y]|n|q)>
```

If `q` is answered, the flow’s execution is terminated as though an `@EXIT`

directive had been encountered. If y is answered (the default), the command block is executed as normal. If n is answered, the loop's execution is terminated as though a @BREAK directive had been encountered.

@SWITCH <test command>
the prompt offers to execute the command or not:

> Execute it? (y|[n]|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered, the command is executed as normal. If n is answered (the default), you are then prompted for the output the command would have produced in order to test the @CASE pattern matching:

> Command reply? (end with @@)>

The output should be terminated with a line containing only the @@ characters. If you do not want to test the pattern matching, enter @@ and you will be prompted for confirmation of the match for each executed @CASE directives.

@CASE [<patterns>]
the prompt indicates if the pattern matching would succeed:

> Matches, confirm command block execution? ([y]|n|q)>

or fail:

> Does not match, execute command block anyways? (y|[n]|q)>

offering to execute the following command block or not.
If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default on success), the command block is executed as normal. If n is answered (the default on failure) the flow is executed as though the @CASE pattern would not match and the next @CASE is block, if any, is tried instead.

@IF <variable> [==|!= <patterns>]
the prompt indicates if the test succeeds:

> Test succeeded, confirm command block execution? ([y]|n|q)>

or fails:

> Test failed, execute command block anyways? (y|[n]|q)>

offering to execute the following command block or not.
If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default on success), the command block is executed as normal. If n is answered (the default on failure) the flow is executed as though the @IF test failed and the @ELSE is block, if any, is tried instead. If this block is executed, the @ELSE block, if any, will be ignored.

@ELSE
the prompt offers to execute the following block:

```
> Confirm command block execution? ([y]|n|q)>
```

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command block is executed as normal. If n is answered, the command block is skipped until the corresponding @END construct.

@INCLUDE <command file>, or
@RUN <command file>
the prompt offers to confirm the execution of the file:

```
> Include/Run the file? ([y]|n|q)>
```

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command file is executed as normal. If n is answered, the command file is not included.

<device command>
the prompt offers to execute the command or not:

```
> Execute it? (y|[n]|q)>
```

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered, the command is executed as normal. If n is answered (the default) the command is not executed and the next command is tried. The behavior is the same for commands prefixed with @TRACE, @NOTRACE, @TRY, and @CRITICAL.

@PRINT <string>
@EXIT [<code>]
@DEFINE <name> <value>
@END
The command is echoed as is without further prompts.

- **NMSCmdSetFlowCBAVL [-iname <name>]**
  <var name> <value> ...
  called within a callback (`NMSCmdBindCallback`). Invoke this command from a command flow executing (`NMSCmdDoCommandFile` or `NMSCmdDoCommandFlow`) by the `@CB` construct. The callback can then inform the flow of the results by setting or resetting some variable assignments. You can specify multiple variable name/value pairs.

- **NMSCmdOpenCommandLog [-iname <name>]**
  [-append]
  <log file path>
  Opens the specified file and logs all commands executed with `NMSCmdSendCommand`, including those issued by a command flow or file. Each device command executed by the flow is logged, not only the flow's execution. If you specify `-append`, the commands are appended to the file if it exists. By default the file is overwritten. The format of the log allows its replay as a command flow or as input to the `cmccmd` utility. The commands are logged from the module on, no group name, and with all variables substituted for flow commands). Note that logging can also be controlled from the flow itself through the `@LOG` and `@LOGFILE` constructs.

- **NMSCmdCloseCommandLog [-iname <name>]**
  Stops command logging and closes the log file opened by `NMSCmdOpenCommandLog`.

Customer Database access

The Customer Database access capabilities or EPI allow you to query and modify the contents of the Customer Databases managed by Preside Multiservice Data Manager (MDM). The Customer Databases are controlled by servers (`cdbserver`) and can be accessed through the Customer Database tool (GUI). Customer Databases can also be accessed by the `cdbextract` and `cdbmerge` utilities, which are more suited for bulk extraction and population. EPI provides more programmability of the database's contents. For example, by using EPI you can receive an alarm feed from GMDR and extract the matching customer information from the database. You can then use this information to generate a trouble ticket to an external system.

Customer Database Access commands are used in the following sequence:

1 Initialize a Cdb interface.

2 Connect to the Cdb server for the appropriate database.

3 Send Fetch, Store, or Erase commands.

4 Send Query commands and receive the matching replies either synchronously with the RecvReply command or asynchronously by binding a script callback to the Cdb interface.

5 Use DtKsh or the provided commands to analyze the Fetch and Query replies.

6 Disconnect from the Cdb server.

7 Drop the Cdb interface.

The Customer Database server is synchronous in that a Cdb interface can only perform one Fetch, Query, Store and Erase command at once. The Fetch, Store, and Erase commands are synchronous in that they always wait for the reply. The Fetch command also provides the matching information on return (similar to a non-wild card Query followed by a RecvReply command).

Customer Database Access commands

The following Customer Database Access commands are provided:

- **NMSCdbInit [-iname <name>]**
  Initializes a Customer Database interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultCdb` to make it the default interface).

- **NMSCdbDrop -iname <name>**
  Drops the specified Customer Database interface (frees allocated resources).

- **NMSCdbConnect [-iname <name>] <CDB name>**
  [<CDB server host>]
  Connects the interface to the CDB server for the indicated database and host (defaults to “localhost”).

  Example

```
NMSCdbInit
if NMSCdbConnect EastCustDB bcarse88
then
   ... # we’re connected
```

- **NMSCdbDisconnect [-iname <name>]**
  Disconnects the interface from the CDB server. The interface may then be reconnected to another server.

- **NMSCdbFetch [-iname <name>] [-out] [-hier] <component ID>**
  Queries the Customer Database for information on the specified component. Typically, the component name is specified in canonical format (see `NMSEPIConvertCompId`). The command blocks and waits for the reply. The reply is returned as environment variables and, if `-out` is specified, on standard output. If `-hier` is specified and no match is found, the routine looks for matching data on the parent components. The `NMSEPI_CDB_COMPID` is also set to the component ID on which the match was found. With `-hier`, both the canonical and display format of the component IDs are searched

  `NMSEPI_CDB_COMPID`
  is the component ID.

  `NMSEPI_CDB_RELCOMPID`
  is the associated component ID, if any; otherwise an empty string

  `NMSEPI_CDB_DATA`
  is the associated textual data.

  `NMSEPI_CDB_DATE`
  is the associated date stamp (as in `YYYYMMDD`).

  `NMSEPI_CDB_SOURCE`
  is the associated three character source code.

  If `-out` is specified, the matching reply is output with the following format (matches the output and input to the `cdbextract` and `cdbmerge` utilities):

  <source:3>;<date:8>;<component ID:65>;<related component ID:46>;<data length:4>;<data: variable>\n

Example

```
if NMSCdbFetch "EM TOTO FRUNI 45"
then
   print "Data: $NMSEPI_CDB_DATA"
   ...
```

If `-out` is specified, the command produces the following output (the text is split with '\' for readability):

```
EPI;19980508;EM TOTO FRUNI 45                \
                                ;FRAD AP34   \
                                 ;0024;Client \
Number: AP0002345
```

- **NMSCdbQuery [-iname <name>]**
  -comp <component ID pattern>
  | rel <component ID pattern>
  | -data <data pattern>

  Queries the Customer Database for information matching the specified grep-style pattern (see `NMSEPIPatternMatch` on page 35). Only one pattern can be specified to match the entry's component name (`-comp`), associated component name (`-rel`), or data (`-data`). The command immediately returns. The returned replies must be extracted synchronously with `NMCdbRecvReply` or asynchronously through a script callback bound to the interface with `NMSCdbBindCallback`.

  Example

  ```
  if NMSCdbQuery -comp "EM TOTO FRUNI .*"
  then
     ... # query sent successfully
  ```

- **NMSCdbRecvReply [-iname <name>] [-out] [<timeout>]**

  Waits for and receives the next Query reply record from the CDB server. A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts as a no-wait poll. If `-out` is specified, the received record is printed on standard output. The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred). The following environment variables are set:

  `NMSEPI_CDB_COMPID`
  is the component ID.

NMSEPI_CDB_RELCOMPID
is the associated component ID, if any; otherwise an empty string.

NMSEPI_CDB_DATA
is the associated textual data.

NMSEPI_CDB_DATE
is the associated date stamp (as in YYYYMMDD

NMSEPI_CDB_SOURCE
is the associated three character source code.

If -out is specified, the matching reply is output with the same format as the NMSCdbFetch command.

NMSCdbRecvReply can be called for each reply and will return 0 (also in NMSEPI_RESULT) to indicate success. It will return 6 to indicate that no more replies are available. Further calls, as well as calls when there is no active Query command, will return an error indication.

Example

```
NMSCdbQuery -data "Client Number: AP.*"
while NMSCdbRecvReply
do
   print "Component: $NMSEPI_CDB_COMPID"
   print "Updated:   $NMSEPI_CDB_DATE"
   print "Data:      $NMSEPI_CDB_DATA"
   ...
```

- **NMSCdbBindCallback [-iname <name>] <callback command>**
  Binds the specified DtKsh command string (typically a function invocation with its arguments) to the Cdb interface. This command string will be executed whenever a Query command reply is received from the server for this interface. The following environment variables are available to the callback:

  NMSEPI_INAME
  is the API interface name of the callback.

  NMSEPI_CB_REASON
  is the reason for the callback (LOST_CONNECTION, ERROR, ENDRESP, or

RESPONSE).

NMSEPI_CDB_COMPID
is the component ID.

NMSEPI_CDB_RELCOMPID
is the associated component ID, if any, an empty string otherwise

NMSEPI_CDB_DATA
is the associated textual data.

NMSEPI_CDB_DATE
is the associated date stamp (as in YYYYMMDD).

NMSEPI_CDB_SOURCE
is the associated three character source code.

Example

```
function mycb {
   # ...Process the reply...
}
...
NMSCdbQuery -data "AP.*"
NMSCdbBindCallback mycb
...
NMSEPIEventLoop # never returns
```

- **NMSCdbStore [-iname <name>] <component ID> <data>**
  [-rel <related component ID>]
  [-date <date as YYYYMMDD>]
  [-source <source as SSS>]

  Stores the specified information in the Customer Database. Typically, the component name is specified in canonical format (see NMSEPIConvertCompId). The information is added, or replaces that already associated with the component. The component name and data must be specified. An associated component name (-rel), a date stamp (-date, defaults to the current date), and a 3-character source code (-source, defaults to EPI) can also be specified. The command is synchronous since it blocks and waits for the reply from the CDB server.

Example

```
if NMSCdbStore "EM TOTO FRUNI 43" "Client Number: DP542
Contact: (613) 763-2211)" -rel "FRAD 213" -source "GCG"
then
   ... # data is now stored
```

- **NMSCdbErase [-iname <name>] <component ID>**
  Discards the information associated with the specified component in the Customer Database. Typically, the component name is specified in canonical format (see `NMSEPIConvertCompId`). The command is synchronous since it blocks and waits for the reply from the CDB server.

  Example

```
if NMSCdbErase "EM TOTO FRUNI 52"
then
   ... # data is now erased
```

Real-Time Alarm Collection

The Real-Time Alarm Collection (RTAC) capabilities of EPI let you query the matching alarms to produce historical alarm reports. RTAC uses the rtaccol server to spool the alarms it retrieves from the GMDR server to the workstation files, one file per day (based on the alarm time stamp). With the EPI RTAC interface, you can query the spooled alarms between two date-time boundaries. You can also specify filters for any alarm attributeby using GREP-style patterns. The matching alarms are retrieved and their attributes are provided in the same way as with the Alarm&Status specialized API interface.

Use RTAC Access commands in the following sequence:

1 Initialize an RTAC interface.

2 Start a query by specifying its date/time boundaries and attribute-value filters.

3 Fetch the matching alarms.

4 For each fetched alarm, extract the desired attributes and process them.

5 Drop the RTAC interface (usually not done).

The RTAC interface is synchronous because an interface can only perform one query at a time and the Fetch command is blocking. You can, however, limit the amount of time and the number of alarms the Fetch command scans for a polling/round-robin form of multi-tasking.

You cannot use the RTAC interface remotely. You must use the RTAC interface on the same machine that stores (or NFS mounts) the RTAC spool files. The spool files are located using the `RTAC.cfg` configuration file. For details, see Real time alarm collection tool (rtaccol) in 241-6001-310 *Preside MDM Server Reference Guide*.

RTAC Access commands

The following RTAC Access commands are available:

- **NMSRTACInit [-iname <name>]**
  Initializes an RTAC interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultRTAC` to make it the default interface).

- **NMSRTACDrop -iname <name>**
  Drops the specified RTAC interface (frees allocated resources).

- **NMSRTACStartQuery [-iname <name>]**
  <start date/time> <end date/time>
  [-filter <attribute name> <pattern> ...]
  Initiates an RTAC query for the alarms within the specified start and end date/time. Only alarms whose attributes match the specified filters will be returned. The start time can be specified as a date ("`yyyy mm dd`"), date and time ("`yyyy mm dd hh mm ss`"), as the special values "`0000 00 00`" or "`ANY`" meaning the oldest available alarm, and as the special value "`NOW`" meaning the current (workstation) date and time. Similarly, the end date/time can be specified as a date ("`yyyy mm dd`"), date and time ("`yyyy mm dd hh mm ss`"), as the special values "`9999 99 99`" or "`ANY`" meaning the latest alarm available, or as the special value "`NOW`", meaning the current (workstation) date and time.
  The filter attribute names are the same as those provided through the Alarm and Status API. For details, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The filter values are specified as GREP style patterns for target values similar to the ones output by the

Alarm&Status API. Multiple attribute-pattern filters can be specified. Filters on the same attribute are ORed together and filters on different attributes are ANDed together (similar to the API rules).

Examples

```
NMSRTACInit
if NMSRTACStartQuery "ANY" "NOW" \
              -filter event   "set" \
                         severity "critical" \
                         severity "major"
then
   ... # ready to fetch all critical/major alarms up
   ... # to now

or,

NMSRTACInit
NMSEPIConvertTime -api -offset -7200
if NMSRTACStartQuery "$NMSEPI_OUTPUT_TIME" "NOW"
then
   ... # ready to fetch alarms for the last two hours
```

To initiate a new query, simply invoke `NMDRTACStartQuery` again.

- **NMSRTACFetchNextAlarm [-iname <name>] [-out]**
  [-terse|-normal|-full]
  [-var <array name>]
  [-timeout <timeout>]
  [-maxrecs <max records>]

  This command fetches the next matching alarm according to the criteria specified by `NMSRTACStartQuery` and `NMSRTACAddFilter`. The matching alarms are returned similarly to the `NMSGMDRAPIRecvAlarm` command. The following environment variables are set;

  `NMSEPI_RECORD_TYPE:`
  is the API record type (always RESPONSE in this context).

  `NMSEPI_ALARM_SEVERITY:`
  is the severity of the alarm.

  `NMSEPI_ALARM_EVENT`
  is the alarm event.

NMSEPI_ALARM_FAULT
is the fault code.

NMSEPI_ALARM_TIME
is the alarm time.

NMSEPI_ALARM_COMPID
is the component ID.

NMSEPI_ALARM_OPDATA
is the operator data.

If you specify the `-out` option, the major attributes are also ouput onto the standard output stream. If you specify `-terse`, `-normal`, or `-full`, then the `NMSEPI_ALARM_DISPLAY` environment variable is set to the corresponding display format (similarly to `NMSGMDRAPIFormatAlarm` and `NMSRTACFormatAlarm`) and output on the standard output stream if `-out` is also specified. If you specify an associative array name with `-var`, the array is fills with all the non-empty attributes of the alarm just like for `NMSGMDRAPIRecvAlarm`.
If you use the `-timeout` and/or `-maxrecs` options, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum number of records tested occurs, the command returns the exit code `5` (timeout) as the `NMSEPI_RESULT` variable. This lets you "poll" the RTAC interface for a while and round-robin to other tasks.

Example

```
if NMSRTACStartQuery "ANY" "NOW" \
               -filter event    "set" \
                       severity "critical" \
                       severity "major"
then
   while NMSRTACFetchNextAlarm -full -var alarm
   do
        if (( ${alarm[compCriticality]} > 60 ))
        then
               print "$NMSEPI_ALARM_DISPLAY"
```

```
        fi
    done
fi
```

If you use the `-timeout` and/or `-maxrecs` options, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum number of records tested occurs, the command returns the exit code `5` (timeout) as the `NMSEPI_RESULT` variable. This lets you "poll" the RTAC interface for a while and round-robin to other tasks.

You can extract the content of the fetched alarm with the commands `NMSRTACFormatAlarm`, `NMSRTACGetAlarm`, `NMSRTACGetFirstAttribute`, `NMSRTACGetNextAttribute`, and `NMSRTACFindNextAttribute`.

- **NMSRTACFormatAlarm [-iname <name>] [-out] terse|normal|full**
  This command produces the last fetched alarm in the specified display format. The output is provided as the `NMSEPI_ALARM_DISPLAY` environment variable and on the standard output stream if the `-out` option is specified.

  Example

```
if NMSRTACStartQuery "ANY" "NOW" \
              -filter event "set" \
                        severity "critical" \
                        severity "major"
then
    while NMSRTACFetchNextAlarm -var alarm
    do
        if (( ${alarm[compCriticality]} > 60 ))
        then
              NMSRTACFormatAlarm full
              print "$NMSEPI_ALARM_DISPLAY"
        fi
    done
```

- **NMSRTACGetAlarm [-iname <name>] [-out] [-var <array name>]**
  This command resets the attribute scan to the beginning of the attribute list (for `NMSRTACGetNextAttribute` and

NMSRTACFindNextAttribute) and produces the last fetched alarm into the named associative array variable (if -var is used) and/or the standard output stream (if -out is used).

- **NMSRTACGetFirstAttribute [-iname <name>] [-out]**
  This extracts the first attribute of the last fetched alarm. The attribute is provided similarly to the NMSAPIGetNextAttribute command.

  NMSEPI_FIELD_LABEL
  is the API field type (always "_attr:" in this context).

  NMSEPI_FIELD_NAME
  is the API attribute name

  NMSEPI_FIELD_TYPE
  is the API attribute type element (always "S" in this context).

  NMSEPI_FIELD_VALUE
  is the API attribute value.

  Multiple-line attribute block values are returned as a single multiple-line value.

  If -out is used, the field is printed in an API-like format on standard output. The return error code is in NMSEPI_RESULT (6 indicates there are no more fields).

- **NMSRTACGetNextAttribute [-iname <name>] [-out]**
  This command is similar to NMSRTACGetFirstAlarm but extracts the next attribute in the list.

  Example

```
NMSRTACGetAlarm
...
while NMSRTACGetNextAttribute
do
   if [[ "$NMSEPI_FIELD_LABEL" = "_attr:" \
       && "$NMSEPI_FIELD_NAME" = "time" ]]
   then
      echo "Time is: $NMSEPI_FIELD_VALUE"
```

- **NMSRTACFindNextAttribute [-iname <name> [-out] <attribute name>**

  This command finds the next named attribute (NRS type name or long name/title) in the attribute list and returns it similarly to `NMSRTACGetFirstAttribute`.

Network Reporting System

The Network Reporting System (NRS) capabilities of EPI let you create device configuration reports on the data stored in the NRS database and its schema (RDF files). You build the NRS query by identifying the node configuration files to scan (by name, pattern, and naming discipline such as keyed and dated)and the component types to extract configuration data from. The configuration data is returned to the EPI script in various ways (for example, environment variable, associative arrays, and standard output). The data values correspond to the way NRS reports currently work. The EPI NRS capabilities also add value by providing program access to the following items:

- NRS schema contents (the RDF files)
- the automatic construction of the component name
- the ability to specify Passport component and attribute types by name and hierarchical path, rather than by numerical IDs.

EPI does not provide a means of populating this database. You must use the NRS utilities for this (some examples include nrspop, pnrspop, and sisauto). For more information on the NRS database and its use, see 241-6001-022 *Preside MDM Network Reporting System User Guide*.

Use the following sequence for NRS Access commands:

1 Initialize an NRS interface.

2 Start a query by specifying the target node configurations and the component types to be reported, and/or

3 Load and interrogate the NRS schema to help create the report (optional).

4 Fetch the matching configuration components. Components are extracted in depth-first order with no garanteed ordering at peer level (as with other NRS reporting mechanisms).

5 For each fetched component, extract the desired attributes and process them.

6 Drop the NRS interface (usually not done).

The NRS interface is synchronous because an interface can only perform one query at a time and the Fetch command is blocking. You can, however, limit the amount of time and the number of alarms the Fetch command scans to support a polling/round-robin form of multi-tasking.

You cannot use the NRS interface remotely. Use the NRS interface on the same machine that stores (or NFS mounts) the NRS database. The database and schema are located using the `NRS.cfg` configuration file. For details, see 241-6001-022 *Preside MDM Network Reporting System User Guide*.

- **NMSNRSInit [-iname <name>]**
  Initializes an NRS interface. If you use multiple interfaces at the same time, specify a name for this interface when you invoke this command. The name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultNRS` to make it the default interface).

- **NMSNRSDrop -iname <name>**
  Drops the specified NRS interface (frees allocated resources).

- **NMSNRSStartQuery [-iname <name>]**
  [[-include] <comp. type>[*|+|^]...]
  [-exclude <comp. type>[*|+|^]...]
  Initiates an NRS query. The query reports the specified component types. Types listed after the `-exclude` option are not reported. By default, no types are reported. Component types are specified as
  `[<device type>/]<component>[*|+|^]`
  where `<device type>` is a recognized NRS device type (`ppc` for Carrier versions of Passports, `ppe` for Enterprise versions, `dpn` for DPN equipment). If no device type is specified, the default type for the installation is taken from the `NRS.cfg` file. The `<component>` specification can be a name for DPN components, a numerical component ID for Passport components, or a name (the full component type name, for example, `FrameRelayUni`, or the prompt, for example, `FrUni`) for Passport components.

Note: The Passport component names are not unique since there can be multiple Passport components with the same name or prompt. Passport component types are uniquely identified by their numeric component ID. Including or excluding component types to the report by name includes or excludes all the possible matches. This may result in unwanted components being reported.

Passport components can also be identified as the full hierarchy of full names and/or abbreviations. For example, to include the `ServiceParameter` component of a Frame Relay UNI Dlci, the following component type can be specified: `ppc/EM-FrUni-Dlci-Sp`. In that case, only the specified Sp subcomponent (FrUni-Dlci-Sp) is returned. `Sp` components of other hierarchies (for example, FrNni-Dlci-Sp) are ignored

The names are not case sensitive. The schema, RDF files, to interpret the names are located as specified in the `NRS.cfg` file.

If the component type specification uses an asterix (`*`) as a suffix, all the component's possible subcomponents (recursively) are included or excluded, as specified. If you use a plus sign (`+`), only its immediate subcomponents are included or excluded. If you use a caret (`^`) as a suffix, then all the component's possible parent components are included or excluded. If the modifier is applied to a full path Passport component specification, then only the related components of the specified full path are added. In non-path specifications, all possible parents are included or excluded. You can combine multiple suffixes. It is possible to include whole sub-hierarchies of components then exclude the subcomponents you don't need by using the `-exclude` option. More component types can be include/excluded with the `NMSNRSReportCompType` command. Specify the `-include` option if you want to include more components after the specification of excluded ones.
To start a new NRS query, call `NMSNRSStartQuery` again.

Example(1)

```
NMSNRSInit
if NMSNRSStartQuery ppc/2 ppc/FrameRelayUni*
then
    ... # ready to specify the source files for an NRS
```

```
    ... # report on the (Carrier) Passport module and
    ... # Frame Relay components (including all its
    ... # subcomponents)
```

Example(2)

```
NMSNRSInit
if NMSNRSStartQuery ppc/EM-AtmIf-Vcc-Vcd^
then
    ... # ready to specify the source files for an NRS
    ... # report on the (Carrier) Atm Virtual Connection
    ... # Description and its indicated parents (Vcc,
    ... # AtmIf and EM).
```

- **NMSNRSReportCompType [-iname <name>]**
  [[-include] <comp. type>[*|+|^]...]
  [-exclude <comp. type>[*|+|^]...]
  | -list [-out]

  Includes or excludes additional component types to the report. See `NMSNRSStartQuery` for the specification of the types. `NMSNRSReportCompType` can be called multiple times to include and exclude component types.

  This command can also be used to list, with the `-list` option, the component types currently included. These are listed, one per line, as the value of the `NMSEPI_OUTPUT_TEXT` environment variable. If `-out` is specified, they are also printed out to the standard output stream.

  Note: The reported component types can be manipulated in the middle of a report by invoking the `NMSNRSReportCompType` command as appropriate.

  Example

```
if NMSNRSStartQuery ppc/2 ppc/FrameRelayUni*
then
   NMSNRSReportCompType -exclude ppc/Signalling \
          ppc/DataLinkConnectionIdentifier*
   ... # Same as previous example except that this time
   ... # the Signalling and DLCI subcomponents (and
   ... # all its subcomponents for the latter) are to
   ... # to be excluded from the report
```

- **NMSNRSAddSource [-iname <name>]**
  -file <file name>
  | -named <device> <module> <name>
  | -keyed <device> <module> <key>
  | -dated <device> <module> <date>
  | -latest <device> <module>
  [<from date/time>]
  |-list [-out]

  This command specifies which NRS data files to report on. The files are located at the path specified in the `NRS.cfg` file. The files are expected to already be there. For details about how to populate the NRS database, see 241-6001-022 *Preside MDM Network Reporting System User Guide*. The file(s) can be specified in a number of ways;

`-file`	Explicitly names the NRS data file to report on (with or without a full path -- the NRS database path as configured in `NRS.cfg` will be used if not specified).
`-named`	Includes NRS data file(s) matching the specified GREP patterns; `<device>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<module>`, the module name, and `<name>`, the configuration file name.
`-keyed`	Includes NRS data file(s) matching the specified GREP patterns; `<device>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<module>`, the module name, and `<key>`, the configuration file key. Keyed configuration file names have a fixed prefix (the key) followed by a variable two-digit counter. `<key>` only matches the key prefix. For a matching key, the file with the highest two-digit suffix is selected.

-dated	Includes NRS data file(s) matching the specified GREP patterns; <device>, the device type (for example, ppc, ppe, dpn), <module>, the module name, and <date>, the configuration file date (six digits, not a pattern). Dated configuration file names have a fixed six-digit prefix (the date) followed by a variable two-digit counter. <date> only matches the date prefix. For Passports,<date> matches the activation date of the NRS data file name. For both Passport and DPN files, the highest dated file up to the specified date is accepted (and the one with the highest two-digit counter suffix).
-latest	Includes NRS data file(s) matching the specified GREP patterns; <device>, the device type (for example, ppc, ppe, dpn), and <module>, the module name. If you specify <fromDateTime> as "YYMMDD", "YYYY MM DD", or "YYYY MM DD HH MM SS", only the NRS data files from that date/time forward (UNIX file modifitation time-stamp) are considered. This option is useful for creating incremental reports based on the latest NRS population or the last report invocation. For each matching module, the matching file with the highest file system date is selected (the most recently populated file).

In all cases where multiple files match the patterns provided in the `NMSNRSAddSource` call, only one file per module is selected, that is, the highest one in alphanumerical order. For Passport, this also means the one with the highest version counter, the three-digit file suffix. Multiple calls to `NMSNRSAddSource` can select multiple files for the same module.

This command can also be used to list, with the `-list` option, the files currently included in the report. They are listed, one per line, with full path, as the value of the `NMSEPI_OUTPUT_TEXT` environment variable. If you specify `-out`, they are also printed out to the standard output stream.

Example(1)

This example and the following assume a sample NRS database containing the following files:

```
dpn.R78.4078.R7872.970709.data
dpn.R78.4078.R7888.970715.data
ppc.NODER16.2105.NEWCARD,full,012.001105.data
ppc.NODER16.2105.NEWCARD05,full,001.001102.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
ppc.NODER17.2106.newconf,full,030.010210.data
ppc.NODER17.2106.demo,full,003.010123.data
ppc.EASTOTT.2100.lab,full,005.010123.data
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
ppe.SOUTHRICH.2102.JF,full,003.010124.data
ppe.SOUTHRTP.2113.voice,full,044.010124.data
```

```
if NMSNRSStartQuery dpn/PE^
then
    NMSNRSAddSource -file \
           “dpn.R78.4078.R7872.970709.data“
    ... # ready to fetch module and PE components from
    ... # the specified NRS data file
```

Given thepreceding sample database, this call selects the following file:

```
dpn.R78.4078.R7872.970709.data
```

Example(2)

```
if NMSNRSStartQuery ppc/Card
then
    NMSNRSAddSource -name "ppc" ".*" "NEWCARD.*"
    ... # ready to fetch card components from all
    ... # node configurations whose name start with
    ... # NEWCARD
```

Given the sample database, this call selects the following files:

```
ppc.NODER16.2105.NEWCARD,full,012.001105.data
```

(the "latest" of the two matching files for NODER16)

```
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
```

Example(3)

```
if NMSNRSStartQuery ppc/Card
then
    NMSNRSAddSource -keyed "ppc" ".*" "NEWCARD"
    ... # similar as above but for strict KEY syntax
```

Given the sample database, this call selects the following files:

```
ppc.NODER16.2105.NEWCARD05,full,001.001102.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
```

(The other NODER16 file was not selected as its name does not match the syntax of a keyed file)

Example(4)

```
if NMSNRSStartQuery ppc/Card
then
    NMSNRSAddSource -dated "ppc" ".*" "010211"
    ... # as above but this time for all node
    ... # configurations dated before or for February
    ... # 11th 2001
```

Given the sample database, this call selects the following files:

```
ppc.NODER16.2105.NEWCARD,full,012.001105.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
ppc.NODER17.2106.newconf,full,030.010210.data
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
```

(All configurations up to 010211 for PPC nodes are taken.)

Example(5)

```
if NMSNRSStartQuery ppc/Software ppe/Software
then
    NMSNRSAddSource -latest "ppc" "EAST.*"
    NMSNRSAddSource -latest "ppe" "SOUTH.*"
    ... # ready to fetch Software components from the
    ... # latest configuration of the nodes whose names
    ... # start with EAST or SOUTH
```

Given the sample database, this call selects the following files:

```
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
ppe.SOUTHRICH.2102.JF,full,003.010124.data
ppe.SOUTHRTP.2113.voice,full,044.010124.data
```

(Assuming the UNIX file data matches the ordering of the activation dates in the file names.)

- **NMSNRSFetchNextComponent [-iname <name>] [-out]**
  [-var <array name> [-byname]]
  [-skip <level>] [-stop <level>]
  [-timeout <timeout>]
  [-maxrecs <maxrecs>]
  [-marker <seek marker>]

  Fetches the next matching component (according to the component types specified in `NMSNRSStartQuery` and `NMSNRSReportCompType`) from the selected (through `NMSNRSAddSource`) NRS data files. The matching component information is available through the following environment variables:

  `NMSEPI_NRS_COMPID`
  is the full Component ID of the fetch component (the component is specified in mixed case, with space separators, for example, "`EM NODER16 FrUni 132 Dlci 206 Sp $`", and can be manipulated with `NMSEPIConvertCompId`).

  `NMSEPI_NRS_COMPTYPE`
  is the component type of the fetched component (the component type is specified as `<device>/<type>` where `<device>` is the NRS device type (`ppc`, `ppe`, or `dpn`), and `<type>` is the component type (a name for DPN, a numerical component ID for Passport, for example `dpn/PE` or `ppe/8664`).

NMSEPI_NRS_COMPTITLE
is the (long) type name of the fetched component., for example ServiceParameterProv for the component identified above.

NMSEPI_NRS_COMPABBREV
is the short type name of the fetched component (for Passport, it is the prompt), for example Sp for the component identified above.

NMSEPI_NRS_VALUE
is the instance value of the fetched component (the value of its most specific component ID category/value pair, for example, "$" for the component identified above).

NMSEPI_NRS_LEVEL
is the hierarhcy level of the fetched component (starting at 0 for the module level, for example 3 for the component identified above). Note that matching components are returned in depth-first order.

NMSEPI_NRS_FILE_PATH
is the NRS data file from which the matching component was fetched.

If you specify the -out option, the first four items above are also output, one per line, to the standard output stream. If you use the -var optionto name an associative array, the array dills with the NRS component description. The array entry index is the attribute type, as specified in NRS, names for DPN and the special values "OAM", "_COMPONENT", "_HIERARCHY_LEVEL", and a numerical attribute ID for Passport components (for example, 16567 for the Frame Relay Sp committedInformationRate attribute). If you specify the -byname option, the index is as indicated except for the special values, which are then reported as "Ownership", "Component" and "Hierarchy_level" respectively, and for Passport components for which the full attribute name is used instead (for example, "committedInformationRate", not "cir"). The entry value is the corresponding attribute value. EPI also includes special entries with _EPI_COMPID, _EPI_TITLE, and _EPI_ABBREV index containing the information corresponding to the NMSEPI_NRS_COMPID, EPI_NRS_COMPTITLE, and EPI_COMPABBREV variables above.

The individual attributes of the fetched component can also be extracted one at a time using the `NMSNRSGetFirstAttribute`, `NMSNRSGetNextAttribute`, and `NMSNRSFindAttribute` commands.

If you use the `-marker` option, the specified parameter must be a component marker previously saved from a call to `NMSNRSGetComponentMarker`. This repositions the NRS file scanning to the specified marker before fetching the next matching component. If the same component type specifications are used, the same component whose marker is saved is returned. See the description of `NMSNRSGetComponentMarker` for more information. If the specified marker is an empty string, the option is ignored (as if it was not specified).

A marker may refer to another file than the one being scanned. In this case, the marked file is pushed onto the interface and the current file is re-scanned from the top when the marked file is fully processed.

If you use the `-skip` option, to specify a hierarchy level, the command returns the next matching component of a hierachy level that is smaller or equal to the one given. Other matching components of a higher level are ignored. This method restricts the search by skipping over components that are found to be undesireable (for example, if a Service Parameter (level 3) component attribute of a Passport Frame Relay DLCI component (level 2) does not meet the necessary criteria, the next call to `NMSCmdFetchNextComponent` can specify `-skip 2` which skips to the next DLCI component, ignoring any intervening matches. NRS components are delivered in depth-first order.

Note: Specifying `-skip 0` is an efficient way of skipping over an entire module (configuration file) since EPI does not to try to match the left over components. The command tries to match components in the next module-configuration specified with `NMSNRSAddSource`. Specifying `-skip -1` has the same effect as not specifying the option at all.

If you use the `-stop` option, the next matching component is returned as normal. In addition, if a non-matching component of the specified level or a lower value is found, the command returns with an empty

component (all variables are empty except for NMSEPI_NRS_LEVEL). As a result, you receive information when the member of a component sub-tree is scanned and another sub-tree of the same level is about to be scanned (the new sub-tree matches, then it is returned as normal, else the empty component is returned as a form of component terminator). For example, assuming FrUnis (level 1) are being extracted with -stop 1 specified as soon as the first one is found, all matching FrUnis will be returned as normal and an empty component is returned once a Vs components (also at level 1 but not requested by the query) is found. This technique allows one to maintain a state machine that knows when not to expect more sub-components of a specific sub-tree.

If you set the -timeout or -maxrecs options, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum number of records tested occurs, the command returns the exit code 5 (timeout) as the NMSEPI_RESULT variable. This lets you "poll" the NRS interface for a while and round-robin to other tasks.

Example(1)

```
# looking for FrUnis with no running LMIs
...
NMSNRSStartQuery ppc/FrUni+
...
skip=-1
while NMSNRSFetchNextComponent -skip $skip \
          -var comp -byname
do
   skip=-1
   if [[   $NMSEPI_NRS_COMPABBREV == "Lmi" \
        && $comp[procedures] == "none" ]]
   then
          print "$NMSEPI_NRS_COMPID has no active Lmi"
   else
          # skip to the next FrUni
          skip=$(( $NMSEPI_NRS_LEVEL - 1 ))
   fi
done
```

Example(2)

```
...
NMSNRSStartQuery ppc/ServiceParametersProv
...
NMSNRSFetchNextComponent -var comp -byname
```

can produce the following values for the comp array:

```
comp[_EPI_COMPID]=\
       ”EM NODER16 FrUni 132 Dlci 206 Sp $”
comp[_EPI_TITLE]=ServiceParametersProv
comp[_EPI_ABBREV]=Sp
comp[Component]=ppe/8664
comp[Hierarchy_level]=3
comp[Ownership]=OWNER_IWS
comp[EM]=NODER16
comp[FrameRelayUni]=132
comp[DataLinkConnectionIdentifier]=206
comp[ServiceParametersProv]=$
comp[committedInformationRate]=64000
comp[measurementInterval]=0
comp[committedBurstSize]=64000
comp[rateEnforcement]=on
comp[rateAdaptation]=off
comp[excessBurstSize]=0
comp[maximumFrameSize]=2100
comp[accounting]=on
comp[updateBCI]=off
comp[raSensitivity]=7
```

without the -byname option, the output is:

```
comp[_EPI_COMPID]=\
       “EM NODER16 FrUni 132 Dlci 206 Sp $”
comp[_EPI_TITLE]=ServiceParametersProv
comp[_EPI_ABBREV]=Sp
comp[_COMPONENT]=ppe/8664
comp[_HIERARCHY_LEVEL]=3
comp[OAM]=OWNER_IWS
comp[_2]=NODER16
comp[_279]=132
comp[_302]=206
comp[_8664]=$
comp[8668]=on
```

```
comp[8670]=64000
comp[8669]=64000
comp[5997]=off
comp[8671]=0
comp[8672]=0
comp[8673]=off
comp[8674]=7
comp[8675]=on
comp[8667]=2100
```

- **NMSNRSGetComponentMarker [-iname <name>] [-out]**
  Extracts a marker for the current fetched component and returns it as the `NMSEPI_NRS_COMP_MARKER` environment variable or on standard output if `-out` is used. The marker may be saved and used later as an argument to `NMSNRSFetchNextComponent's -marker` option to rescan a component structure (not the whole file) which works around the NRS peer component type that is not specified (all instances of a subcomponent type X are together, but it is not specified if instances of type X appear before or after those of its peer type Y). For example, an NRS interface may be used to scan for Frame Relay interfaces and their DNA sub-components. When an FrUni is found, its marker is saved and the interface is used to locate its Dna. Then the same (requires one to manipulate the selected types with `NMSNRSReportCompType`) or another NRS interface can be used to scan for the FrUni's Dlci subcomponents. The marker may be exchanged between properly initialized NRS interfaces. markers may also refer to different files. If the marker cannot be computed, an empty string is returned.

  Example

```
# Assume two NRS interfaces (inames fruni and dlci)
# configured to scan for FrUnis/Dnas and Dlcis
# respectively
while NMSNRSFetchNextComponent -iname fruni -var comp
do
   case $NMSEPI_NRS_COMP_ABBREV in
   FrUni )
         # ... process FrUni information...
         NMSNRSGetComponentMarker
         marker="$NMSEPI_NRS_COMP_MARKER"
         ;;
   Dna )
```

```
            # ... process DNA information ...
            stopAt=-1
            while NMSNRSFetchNextComponent \
                          -iname dlci -stop $stopAt \
                          -marker "$marker"
            do
               marker=""
               stopAt=1
               # ... process the DLCIs ...
            done
            ;;
   esac
done
```

- **NMSNRSGetComponent [-iname <name>] [-out] [-var <array name> [-byname]]**

  Resets the fetched component's attribute list for the `NMSNRSGetFirstAttribute`, `NMSNRSGetNextAttribute`, and `NMSNRSFindAttribute` commands and returns the description of the component similarly to `NMSNRSFetchNextComponent`.

- **NMSNRSGetFirstAttribute [-iname <name>] [-out] [-var <array name>]**

  Extracts the first attribute from the fetched component. The following environment variables are set with the description of the attribute:

  `NMSEPI_FIELD_LABEL`
  is the attribute's name (for Passport, it is the attribute's full name, not its prompt, for example, "`Percent heap`" for DPN, "`committedInformationRate`" for Passport).

  `NMSEPI_FIELD_NAME`
  is the attribute's type; a string for DPN, a numeric attribute ID for Passport (for example, "`LOADPETYPE`" for DPN, `8669` for Passport).

  `NMSEPI_FIELD_TYPE`
  is the NRS type of the attribute; `BIT_STRING`, `BOOLEAN`, `DNA`, `HEXADECIMAL`, `INVISIBLE`, `LISTINDEX`, `NOKEY`, `NUMERIC`, or `STRING`.

  `NMSEPI_FIELD_VALUE`
  is the attribute's value.

If you specify the `-out` option, the values also print to the standard output stream as one line in the same order as above.
If you use the `-var` option to name an associative array, that array is filled with the attribute's description with entries with the following indicies:

`name`
the attribute's type (same as `NMSEPI_FIELD_NAME` above).

`value`
the attribute's value (same as `NMSEPI_FIELD_VALUE` above).

`type`
the attribute's NRS type (same as `NMSEPI_FIELD_TYPE` above).

`title`
the attribute's full name (same as `NMEPI_FIELD_LABEL` above).

`abbrev`
for DPN, it is the same as the `title` entry, for Passport it is the attribute's prompt.

`width`
the attribute's maximum width in NRS.

- **NMSNRSGetFirstAttribute [-iname <name>] [-out] [-var <array name>]**
  Extracts the next attribute from the fetched component. The attribute information is returned similarly to `NMSNRSGetFirstAttribute`.

- **NMSNRSFindAttribute [-iname <name>] [-out] [-var <array name>] <name>**
  Extracts the named attribute from the fetched component. The search is case-insensitive. For Passport, both the full attribute name and the prompt can be used. The attribute information is returned in a simila manner to `NMSNRSGetFirstAttribute`.

Example

```
# looking for FrUnis with no running LMIs (as above)
...
NMSNRSStartQuery ppc/EM-FrUni-Lmi
...
skip=-1
while NMSNRSFetchNextComponent -skip $skip
do
    skip=-1
    NMSNRSFindAttribute "procedures"
    if [[ $NMSEPI_FIELD_VALUE == "none" ]]
    then
          print "$NMSEPI_NRS_COMPID has no active Lmi"
    else
          # skip to the next FrUni
          skip=$(( $NMSEPI_NRS_LEVEL - 1 ))
    fi
done
```

- **NMSNRSGetComponentSchema [-iname <name>] [-out] [-var <array name> [-byname]] [<comp type>[*|+|^]]**

  Use this command to examine the component schema for the current fetched component (from `NMSNRSFetchNextComponent`) when no `<comp type>` is specified. You can also use it to examine the NRS schema (RDFs) in general when `<comp type>` is specified in a similar manner to the `NMSNRSStartQuery` command. Use the `*`, `+`, or `^` prefix to automatically load the RDFs for the subcomponents and parents of the specified component respectively. (Note that components loaded with this command are not necessarily reported. Use `NMSNRSStartQuery` or `NMSNRSReportCompType` for this).

  Note: For Passport, `<comp type>` can also be specified as the component's numerical ID, name, prompt or full name/prompt path. Passport component names and prompts are not unique since there can be multiple Passport components with the same name or prompt (but different numerical IDs). It is not determined which matching component will be returned by `NMSNRSGetComponentSchema` in that case.

  The following environment variables are set by this command:

NMSEPI_NRS_SCHEMA_NAME
is the component's type specified as <device>/<comp> where <device> is the device type (ppc, ppe, or dpn), and <comp> is the component type (a name for DPN, a numerical ID for Passport, for example, dpn/UTP for DPN, ppc/302 for Passport).

NMSEPI_NRS_SCHEMA_TITLE
is the component's name (a name for DPN, the full component name, for Passport, for example, "UTP" for DPN, "FrameRelayUni" for Passport).

NMSEPI_NRS_SCHEMA_ABBREV
is the component's abbreviation (a name for DPN, the prompt for Passport, for example, "UTP" for DPN, "FrUni" for Passport).

NMSEPI_NRS_SCHEMA_OWNERS
is the list of the component's direct parent component types. The types are on a single line, space separated, and specified as <device>/<comp> similarly to NMSEPI_NRS_SCHEMA_NAME above. For example, the owners for dpn/UTP are "dpn/PO", and for ppc/VritualFramer they are "ppc/9200 ppc/1543 ppc/279 ppc/3742".

NMSEPI_NRS_SCHEMA_SUBORDINATES
is the list of the component's direct sub-component types. The types are on a single line, space separated, and specified as <device>/<comp> similarly to NMSEPI_NRS_SCHEMA_NAME above. For example, for dpn/UTP the subordinates are "dpn/UTPLINK dpn/UTPLINK_EXT dpn/MI8_CUS_MODEM_PROF_ENV dpn/UTPDIALUP dpn/UTPPASSWD_ENV" , and for ppc/DataLinkConnectionIdentifier they are "ppc/8664 ppc/219 ppc/16565".

NMSEPI_NRS_SCHEMA_ATTRS
is the list of the component's attribute types. The types are on a single line, separated by spaces, and specified as names for DPN and as numerical IDs for Passport. For example, for dpn/UTP,the attributes are "_COMPONENT _HIERARCHY_LEVEL _PM _PE _PI _PO _UTP OAM", and for ppc/FrameRelayUni they are "_COMPONENT _HIERARCHY_LEVEL _2 _279 OAM 96 3177 3175 9332".

If you specify the `-out option`, the component type, (full) name, abbreviation (prompt), list of direct parent component types, list of direct subcomponent types, and list of attributes are printed to the standard output stream, one per line (the last two lists are on a line each, with space separators between the types).

If you use the `-var` option to name an associative array, the command fills two arrays with the schema information. The `<array name>` array contains the following entries indicated by their index:

`name`
is the component's type (same as `NMSEPI_NRS_SCHEMA_NAME` above).

`title`
is the component's name (same as `NMSEPI_NRS_SCHEMA_TITLE` above).

`abbrev`
is the component's abbreviation (same as `NMSEPI_NRS_SCHEMA_ABBREV` above).

`owners`
is the component's list of direct parent types (same as `NMSEPI_NRS_SCHEMA_OWNERS` above).

`subordinates`
is the component's list of direct subcomponent types (same as `NMSEPI_NRS_SCHEMA_SUBORDINATES` above).

`attrs`
is the component's attribute types (same as `NMSEPI_NRS_SCHEMA_ATTRS` above).

The `<array name>`_fields array contains the following entries for each attribute type supported by the component (where `<field type>` is the attribute name for DPN and the attribute numeric ID for Passport or the attribute full name/title if `-byname` is specified.

`<field type>,`**`name`**

is the indexed field NRS attribute type name.

`<field type>,`**`title`**
is the indexed field name (a string for DPN, the full attribute name for Passport).

`<field type>,`**`width`**
is the indexed field maximum NRS width.

`<field type>,`**`abbrev`**
is the indexed field abbreviation (same as the title for DPN, the attribute prompt for Passport).

`<field type>,`**`group`**
is the indexed field attribute group type.

Example(1)

```
# Lists (by full name) the attributes of the Passport
# Lp component
NMSNRSGetComponentSchema -var comps \
          ppc/LogicalProcessor
print "$NMSEPI_NRS_SCHEMA_TITLE"
# print its attributes by name
for i in $NMSEPI_NRS_SCHEMA_ATTRS
do
   print " ${comps_fields[$i,title]}"
done
```

Example(2)

```
NMSNRSInit
NMSNRSGetComponentSchema -var comp dpn/UTP
```

can produce the following values for the `comp` array:

```
comp[name]=dpn/UTP
comp[title]=UTP
comp[abbrev]=UTP
comp[owners]=dpn/PO
comp[subordinates]="dpn/UTPLINK dpn/UTPLINK_EXT \
        dpn/MI8_CUS_MODEM_PROF_ENV dpn/UTPDIALUP \
        dpn/UTPPASSWD_ENV"
comp[attrs]="_COMPONENT _HIERARCHY_LEVEL _PM _PE \
```

```
_PI _PO _UTP OAM"
```

and the following values for the `comp_fields` array:

```
comp_fields[_COMPONENT,abbrev]=Component
comp_fields[_COMPONENT,name]=_COMPONENT
comp_fields[_COMPONENT,title]=Component
comp_fields[_COMPONENT,type]=STRING
comp_fields[_COMPONENT,width]=0
comp_fields[OAM,abbrev]=Ownership
comp_fields[OAM,name]=OAM
comp_fields[OAM,title]=Ownership
comp_fields[OAM,type]=STRING
comp_fields[OAM,width]=10
comp_fields[_HIERARCHY_LEVEL,abbrev]=\
        Hierarchy_level
comp_fields[_HIERARCHY_LEVEL,name]=\
        _HIERARCHY_LEVEL
comp_fields[_HIERARCHY_LEVEL,title]=Hierarchy_level
comp_fields[_HIERARCHY_LEVEL,type]=NUMERIC
comp_fields[_HIERARCHY_LEVEL,width]=2
comp_fields[_PM,abbrev]=PM
comp_fields[_PM,name]=PM
comp_fields[_PM,title]=PM
comp_fields[_PM,type]=STRING
comp_fields[_PM,width]=12
comp_fields[_PE,abbrev]=PE
comp_fields[_PE,name]=PE
comp_fields[_PE,title]=PE
comp_fields[_PE,type]=NUMERIC
comp_fields[_PE,width]=2
comp_fields[_PI,abbrev]=PI
comp_fields[_PI,name]=PI
comp_fields[_PI,title]=PI
comp_fields[_PI,type]=NUMERIC
comp_fields[_PI,width]=2
comp_fields[_PO,abbrev]=PO
comp_fields[_PO,name]=PO
comp_fields[_PO,width]=2
comp_fields[_PO,type]=NUMERIC
comp_fields[_PO,title]=PO
comp_fields[_UTP,abbrev]=UTP
comp_fields[_UTP,name]=UTP
```

```
comp_fields[_UTP,title]=UTP
comp_fields[_UTP,type]=NOKEY
comp_fields[_UTP,width]=1
```

Example(3)

```
NMSNRSInit
NMSNRSGetComponentSchema -var comp ppc/FrameRelayUni
```

can produce the following values for the `comp` array:

```
comp[name]=ppc/279
comp[title]=FrameRelayUni
comp[abbrev]=FrUni
comp[owners]=ppc/2
comp[subordinates]="ppc/161 ppc/16502 ppc/287 \
        ppc/10990 ppc/586 ppc/9314 ppc/302 \
        ppc/10649 ppc/8600 ppc/1768"
comp[attrs]="_COMPONENT _HIERARCHY_LEVEL _2 _279 \
        OAM 96 3177 3175 9332"
```

and the following values for the `comp_fields` array:

```
comp_fields[_COMPONENT,abbrev]=Component
comp_fields[_COMPONENT,group]=
comp_fields[_COMPONENT,name]=_COMPONENT
comp_fields[_COMPONENT,title]=Component
comp_fields[_COMPONENT,type]=STRING
comp_fields[_COMPONENT,width]=0
comp_fields[OAM,abbrev]=Ownership
comp_fields[OAM,group]=
comp_fields[OAM,name]=OAM
comp_fields[OAM,title]=Ownership
comp_fields[OAM,type]=STRING
comp_fields[OAM,width]=10
comp_fields[_HIERARCHY_LEVEL,abbrev]=\
        Hierarchy_level
comp_fields[_HIERARCHY_LEVEL,group]=
comp_fields[_HIERARCHY_LEVEL,name]=\
        _HIERARCHY_LEVEL
comp_fields[_HIERARCHY_LEVEL,title]=Hierarchy_level
comp_fields[_HIERARCHY_LEVEL,type]=NUMERIC
comp_fields[_HIERARCHY_LEVEL,width]=2
comp_fields[_2,abbrev]=EM
comp_fields[_2,group]=
```

```
comp_fields[_2,name]=_2
comp_fields[_2,title]=EM
comp_fields[_2,type]=STRING
comp_fields[_2,width]=12
comp_fields[_279,abbrev]=FrUni
comp_fields[_279,group]=
comp_fields[_279,name]=_279
comp_fields[_279,title]=FrameRelayUni
comp_fields[_279,type]=NUMERIC
comp_fields[_279,width]=5
comp_fields[3175,abbrev]=ifI
comp_fields[3175,group]=
comp_fields[3175,name]=3175
comp_fields[3175,group]=IfEntryProv
comp_fields[3175,title]=ifIndex
comp_fields[3175,type]=NUMERIC
comp_fields[3175,width]=5
comp_fields[3177,abbrev]=ifState
comp_fields[3177,group]=
comp_fields[3177,name]=3177
comp_fields[3177,group]=IfEntryProv
comp_fields[3177,title]=ifAdminStatus
comp_fields[3177,type]=STRING
comp_fields[3177,width]=7
comp_fields[96,abbrev]=cid
comp_fields[96,group]=
comp_fields[96,name]=96
comp_fields[96,group]=CustomerIdentifierData
comp_fields[96,title]=customerIdentifier
comp_fields[96,type]=STRING
comp_fields[96,width]=4
comp_fields[9332,abbrev]=numberOfEmissionQs
comp_fields[9332,group]=
comp_fields[9332,name]=9332
comp_fields[9332,group]=EmissionPriorityQs
comp_fields[9332,title]=numberOfEmissionQs
comp_fields[9332,type]=STRING
comp_fields[9332,width]=1
```

If -byname had been specified, the fields would have been reported as follows instead:

```
comp_fields[Component,abbrev]=Component
comp_fields[Component,group]=
```

```
comp_fields[Component,title]=Component
comp_fields[Component,type]=STRING
comp_fields[Component,width]=0
comp_fields[Ownership,abbrev]=Ownership
comp_fields[Ownership,group]=
comp_fields[Ownership,title]=Ownership
comp_fields[Ownership,type]=STRING
comp_fields[Ownership,width]=10
comp_fields[Hierarchy_level,abbrev]=\
       Hierarchy_level
comp_fields[Hierarchy_level,group]=
comp_fields[Hierarchy_level,title]=Hierarchy_level
comp_fields[Hierarchy_level,type]=NUMERIC
comp_fields[Hierarchy_level,width]=2
comp_fields[EM,abbrev]=EM
comp_fields[EM,group]=
comp_fields[EM,name]=_2
comp_fields[EM,title]=EM
comp_fields[EM,type]=STRING
comp_fields[EM,width]=12
comp_fields[FrameRelayUni,abbrev]=FrUni
comp_fields[FrameRelayUni,group]=
comp_fields[FrameRelayUni,name]=_279
comp_fields[FrameRelayUni,title]=FrameRelayUni
comp_fields[FrameRelayUni,type]=NUMERIC
comp_fields[FrameRelayUni,width]=5
comp_fields[ifIndex,abbrev]=ifI
comp_fields[ifIndex,group]=IfEntryProv
comp_fields[ifIndex,name]=3175
comp_fields[ifIndex,title]=ifIndex
comp_fields[ifIndex,type]=NUMERIC
comp_fields[ifIndex,width]=5
comp_fields[ifAdminStatus,abbrev]=ifState
comp_fields[ifAdminStatus,group]=IfEntryProv
comp_fields[ifAdminStatus,name]=3177
comp_fields[ifAdminStatus,title]=ifAdminStatus
comp_fields[ifAdminStatus,type]=STRING
comp_fields[ifAdminStatus,width]=7
comp_fields[customerIdentifier,abbrev]=cid
comp_fields[customerIdentifier,name]=96
comp_fields[customerIdentifier,group]=\
       CustomerIdentifierData
comp_fields[customerIdentifier,title]=\
       customerIdentifier
```

```
comp_fields[customerIdentifier,type]=STRING
comp_fields[customerIdentifier,width]=4
comp_fields[numberOfEmissionQs,abbrev]=\
        numberOfEmissionQs
comp_fields[numberOfEmissionQs,name]=9332
comp_fields[numberOfEmissionQs,group]=\
        EmissionPriorityQs
comp_fields[numberOfEmissionQs,title]=\
        numberOfEmissionQs
comp_fields[numberOfEmissionQs,type]=STRING
comp_fields[numberOfEmissionQs,width]=1
```

Sample DtKsh scripts

This section provides the following three sample DtKsh EPI scripts:

- “DtKsh alarm display sample” (page 126) (API interface)
- “Passport Card inventory sample” (page 129) (Command interface)
- “Passport DLCI Configuration Report sample” (page 133) (NRS interface)

DtKsh alarm display sample

The following is a simple graphical interface for an alarm display in DtKsh. It creates a main Motif window with a File menu (Exit command) and a scrolled text field to display the alarms. No real formatting of the alarm is performed, other than to convert the API canonical component ID into display format. This example is available in the Preside Multiservice Data Manager (MDM) load in `/opt/MagellanNMS/cfg/macros/nms/src/AlarmDisplay.dtksh`.

```
#!/usr/dt/bin/dtksh

# load the DtKsh EPI extensions

. /opt/MagellanNMS/lib/nmsepi.ksh


# This callback function is invoked to display the
# alarm in the scrolled text window.

function almCB {

   if [ “$NMSEPI_RECORD_TYPE” != “ALARM” ]
```

```
    then
        exit 1
    fi

    # Convert the component to display format and
    # output the alarm.
    NMSEPIConvertCompId -disp "$NMSEPI_ALARM_COMPID"
    XmTextGetLastPosition pos ${REPTEXT_W}
    XmTextInsert ${REPTEXT_W} $pos "
$NMSEPI_ALARM_EVENT $NMSEPI_ALARM_SEVERITY
$NMSEPI_ALARM_FAULT $NMSEPI_ALARM_TIME
$NMSEPI_COMP_ID
"
}

# Initialize the X-window environment and create the
# Top Level 'window'. This must be done first before
# any DtKsh EPI Initialization calls for GUIs.
XtInitialize TOPLEVEL DtKAlarm "DtkAlarm" \
"DtkAlarm" "$@" title:"Alarm Display" \
iconName: "Alarm Display"

# Create the main window.
XmCreateMainWindow MAINW_W ${TOPLEVEL} "mainW"

# Create the Menu bar, File menu, and Exit button.
XmCreateMenuBar MB_W ${MAINW_W} "menuB"
XmCreatePulldownMenu FILE_M ${TOPLEVEL} "fileMenu"
XmCreatePushButton FILE_M.EXIT ${FILE_M} "exitCmd" \
    labelString:"Exit"      mnemonic:x \
    accelerator:"Ctrl<Key>E" acceleratorText:"Ctrl+E"\
    activateCallback:"exit 0"
XtManageChild ${FILE_M.EXIT}
```

```
XmCreateCascadeButton MB_W.FILE_C ${MB_W} “exitCB” \
    subMenuId:”$FILE_M” labelString:File \
    mnemonic:F
XtManageChild ${MB_W.FILE_C}
XtManageChild ${MB_W}

# Create the Scrolled Text panel.
XmCreateScrolledText REPTEXT_W ${MAINW_W} “scText” \
    editMode:MULTI_LINE_EDIT  columns:80  rows:24
XtManageChild ${REPTEXT_W}

# Complete the GUI by managing and realizing the
# application.
XtManageChild ${MAINW_W}
XtRealizeWidget ${TOPLEVEL}

# Initialize the Alarm and Status API interface.
NMSGMDRAPIInit

# Connect to the Service Selected GMDR server.
NMSGMDRAPIConnect -ssel

# Perform an API REGISTER command sequence.
NMSAPIRegister adisp

# Create the default (simple) alarm sieve with no
# filters.
NMSGMDRAPICreateAlarmSieve

# Bind the callback to be invoked with each received
# alarm.
NMSAPIBindCallback almCB
```

```
# Jump in the event loop (never returns) to handle GUI
# and API events.

XtMainLoop
```

Passport Card inventory sample

The following DtKsh sample script uses the command interface to produce a card inventory of a specified Passport node (using the current User Session). This sample is available in the Preside Multiservice Data Manager (MDM) load in `/opt/MagellanNMS/cfg/macros/nms/src/PPCardInv.dtksh`.

```
#!/usr/dt/bin/dtksh

# load the DtKsh EPI extensions

. /opt/MagellanNMS/lib/nmsepi.ksh


# Extract the arguments.

if [ $# -lt 2 ]

then

    echo "ppcardrep <group> <passport>"

    exit 1

fi

grp=$1

mod=$2


# Initialize the command interface.

NMSCmdInit

NMSCmdConnect

if [ $? != 0 ]

then

    echo "Failed to connect to the Command Session \
    servers."

    exit 1
```

```
fi

echo "
                    Passport Card Inventory
-------------------------------------------------\
---------------------
Node       Card Type       Inserted    Serial #     \
Firm. Rev.     LP
---------- ---- ----------- ---------- -------------\
------------- ------"

# Do we have access to the node.
NMSCmdSendCommand $grp "$mod h -v(d) shelf card"
NMSCmdRecvFullReply
NMSCmdPatternMatch "Shelf Card"
if [ $? != 0 ]
then
   echo "
*** Passport node $mod does not seem to be reachable.
"
   exit 1
fi
if NMSCmdPatternMatch "noTabular"
then
   opt="-noTabular"
fi

# Get the number of slots from the shelf component.
NMSCmdSendCommand $grp "$mod d $opt shelf \
numberOfSlots";
```

```
while NMSCmdRecvNextLine
do
    if NMSCmdGetColumn 1 “numberOfSlots”
    then
        NMSCmdGetColumn 3; \
        numberOfSlots=”$NMSEPI_OUTPUT_COLUMN”
    fi
done

# Get the needed attributes from all card components.
NMSCmdSendCommand $grp “$mod d $opt shelf card/* \
cardType,insertedCardType,serialNumber,\
activeFirmwareVersion,currentLP”
while NMSCmdRecvNextLine
do
    NMSCmdGetColumn 1; col=”$NMSEPI_OUTPUT_COLUMN”
    NMSCmdGetColumn 3; val=”$NMSEPI_OUTPUT_COLUMN”
    case “$col” in
    “cardType” )cardType=”$val” ;;
    “insertedCardType”) insertedCardType=”$val” ;;
    “activeFirmwareVersion” ) \
    activeFirmwareVersion=”$val” ;;
    “serialNumber”) serialNumber=”$val” ;;
    “currentLP”)  currentLP=”$val” ;;
    “Shelf”   )
        # This is the name of a card (either the first
        # or another one in the list.
        if [[  ! -z “$cardType”                    \
            && (  “$cardType”        != “none”   \
                || “$insertedCardType” != “none” ) ]]
```

```
        then
            # Print the information on the preceding
            # card.
            printf \
            "%-12s %-4s %-12s %-12s %-14s %-14s
            %s\n" "$mod" "$card" "$cardType" \
            "$insertedCardType" "$serialNumber" \
            "$activeFirmwareVersion" "$currentLP"
    cardType=""
    fi

    # Extract the card number from the name and stop if
    # maximum.
    NMSCmdGetColumn 2
    NMSEPIPatternMatch "Card/" "$NMSEPI_OUTPUT_COLUMN"
    ""
        card="$NMSEPI_OUTPUT_MATCH"
        if (( $card > $numberOfSlots ))
        then
            break
        fi
        ;;
    esac
    done

    # Print the last card's information if required.
    if [[  ! -z "$cardType"                    \
        && (  "$cardType"         != "none"   \
            || "$insertedCardType" != "none" ) ]]
    then
```

```
    printf "%-12s %-4s %-12s %-12s %-14s %-14s %s\n" \
       "$mod" "$card" "$cardType" "$insertedCardType" \
        "$serialNumber" "$activeFirmwareVersion" \
         "$currentLP"
    fi

    exit 0
```

Passport DLCI Configuration Report sample

The following DtKsh script uses the NRS interface to produce a produce a simple report all configured Frame Relay DLCIs in the network and their major quality-of-service parameters. This example is available in the Preside Multiservice Data Manager (MDM) load in `/opt/MagellanNMS/cfg/macros/nms/src/DlciScan.ksh`.

```
#!/usr/dt/bin/dtksh

# Load the NMS EPI extensions for reporting

. /opt/MagellanNMS/lib/nmsepi.ksh

. /opt/MagellanNMS/lib/nmsepir.ksh


# Initialize an NRS interface.

NMSNRSInit


# Initiate a query for the various FrameRelay Service
# Parameter components.

NMSNRSStartQuery ppc/FrameRelayUni \
                 ppc/EM-FrUni-Dlci-Sp


# Report on all latest configurations (file system
# date) for all nodes.

NMSNRSAddSource -latest "ppc" ".*"
```

```
# Scan the NRS database for all matching components
while NMSNRSFetchNextComponent -var comp -byname
do
    # Extract the major fields by name (the associative
    # array has them by type)
    if [[ $NMSEPI_NRS_COMPABBREV == FrUni ]]
    then
        cid="${comp[customerIdentifier]}"
        continue
    else
        cir="${comp[committedInformationRate]}"
        bc="${comp[committedBurstSize]}"
        be="${comp[excessBurstSize]}"
    fi
    # Print the component information in
    # comma-separated format for the target system
    NMSEPIConvertCompId -disp "$NMSEPI_NRS_COMPID"
    print  "$NMSEPI_COMP_ID,${cid},${cir},${bc},${be}"
done
```

Output redirection and piping

Shell languages, such as DtKsh, support output redirection and piping. This allows you to combine multiple programs and utilities to achieve the end result (in the EPI context, to parse, filter, and transform the output of queries). However, this is typically implemented by spawning the members of the pipes into different independent but communicating (through standard I/O) sub-processes. For EPI, this means that the effect of one of the sub-processes on the internal state of the EPI may not be known to the other processes or to the main Shell script itself.

This usually occurs when a query is started in the Shell and then it receives the replies, piping the results to another process.

Example

```
NMSCmdSendCommand myGroup “dir”
NMSCmdRecvFullReply -out | grep “UP”
```

The `NMSCmdRecvFullReply` call, which receives all responses to the end of the reply, is executed in a separate sub-process. The main Shell script is therefore unaware that the command has been completed, and so it refuses to issue any other queries through its EPI interface. To avoid this kind of piping, use the EPI environment variables:

Example

```
NMSCmdSendCommand myGroup “dir”
NMSCmdRecvFullReply
echo “$NMSEPI_OUTPUT_TEXT” | grep “UP”
```

Alternatively, ensure that the sending of queries and receiving of replies are performed in the same sub-process:

Example

```
{ NMSCmdSendCommand myGroup “dir”; NMSCmdRecvFullReply
-out; } \
| grep “UP”
```

You can also use the EPI commands to achieve the same result.

Example

```
NMSCmdSendCommand myGroup “dir”
while NMSCmdRecvNextLine
do
   NMSCmdPatternMatch “UP” && echo
“$NMSEPI_OUTPUT_TEXT”
done
```

Note: This problem also occurs when using back quotes (‘‘) to convert the output of a command to a string.

Chapter 3
Tcl Embedded Programming Interface

This section describes the Embedded Programming Interface (EPI) in the Tool Command Language (Tcl) scripting language. This chapter contains the following:

Code conventions

There are two code conventions used in this chapter:

- \
  A back slash (\) indicates that the line of code is continued on the next line space.
- #
  A message line that starts with an octothorp (#) is treated as a comment.

Integration methodology

Tcl scripts can be run with a number of interpreters in various locations, depending on how they are compiled and installed. These include *tclsh* which is the plain Tcl Shell, *wish* which combines plain Tcl and graphical Tk extensions, *expect* which combines Expect extensions and Tcl, and *expectk* which combines Expect and Tk with Tcl.

Tcl Shells can be extended by adding new built-in commands. To add new built-in commands to the Tcl, the only requirement is to define them along with a special initialization routine in a dynamic shared library. The library is loaded as follows:

load **<shared library path> <library name>**

where:

`shared library path` is the extension library path.

`library name` is the extension package name (`Epitcl`).

> ***Note:*** Tcl EPI needs Tcl version 7.5 or above. Use “info tclversion” to check the version of your Tcl interpreter.

The Tcl EPI extension library (`libEPITcl.so`) contains all of the required implementation functions for the base, API, Command, Customer Database, and Miscellaneous interfaces. It also contains references to all of the other Preside Multiservice Data Manager (MDM) libraries it depends on (for example, `libIPI.so`, `libAPI.so`, and `libCom.so`). Therefore, to access the preceding interfaces, you only need to load and specify one library in the load command. Other libraries load automatically. Similarly, `libEPITclR.so` contains the implementation of the RTAC and NRS

interfaces.This means only one library needs to be loaded and specified in the load command to access the interfaces listed above; the others are loaded automatically. For example:

```
load /opt/MagellanNMS/lib/libEPITcl.so EpiTcl;
```

The `load` command call is performed by an Preside Multiservice Data Manager (MDM) module, `nmsepi.tcl`. The Tcl EPI scripts must load this module as follows to access the basic interfaces listed above:

```
source /opt/MagellanNMS/lib/nmsepi.tcl;
```

To access the RTAC and NRS interfaces, the following script can be sourced:

```
source /opt/MagellanNMS/lib/nmsepir.tcl;
```

Both sets of interfaces can be loaded by sourcing the two files.

Interface

Tcl EPI provides five groups of built-in commands:

- **Base**:
  Provides mapping to the EPI base routines and utilities as well as additional housekeeping routines.
- **Generic API Access**:
  Provides access to the Generic Application Programming Interface (API) commands and replies.
- **Specialized API Access**:
  Provides specialized and value-added access to specific API Providers such as the Alarm and Status, Network Model, and Host Group Directory Service (HGDS) APIs.
- **Command Access**:
  Provides access to the Preside Multiservice Data Manager (MDM) command macro capabilities.
- **Real-Time Alarm Collection (RTAC)**
  Provides access to the spooled alarms collected by RTAC.
- **Network Reportinng System (NRS)**
  Provides access to the network-wide configutation data collected by NRS.

- **Customer Database Access:**
  Provides access to the Preside Multiservice Data Manager (MDM) Customer Database capabilities.

Command usage information

The following information applies to built-in commands:

- **help**
  All commands support a -h (help) option that displays arguments to a command
- **command argument**
  You need to specify all command arguments in the indicated order.
- **unnamed and named connections**
  By default, the connection-oriented commands (API or Command Access) operate on a single unnamed connection. It is possible, however, to create concurrent named connections. To do so, specify the connection name as an argument (`-iname`) to a command (except for the `NMSEPIDefaultAPI`, `NMSEPIDefaultCmd`, `NMSEPIDefaultCdb`, `NMSEPIDefaultRTAC`, or `NMSEPIDefaultNRS` commands).
- **error output**
  Error messages are usually output to the standard error stream. You can stop output to the standard error stream by using the NMSEPISet command.
- **return codes**
  Commands usually return `1` upon success and `0` upon failure. The global variable `NMSEPI_RESULT` stores the reason for failure. The possible values and the corresponding meanings for `NMSEPI_RESULT` are as follows:

0	success
1	error in argument list
2	default/named interface not created/initialized
3	default/named interface not connected to server
4	operation failed
5	operation timed out

6 no more replies

7 attempt to create an exiting interface (name)

This differs from the DtKsh EPI where commands return the error code. This difference allows you to use EPI commands in Tcl flow control structures, where `1` indicates success (in DtKsh and most other Shells, `0` indicates success).

Example

```
if { [ NMSGMDRAPIConnect -ssel ] } {
... # successfully connected
}
```

Some commands also support an `-out` argument. This argument forces EPI to return its (string) result instead of the return codes.

- **global variables**
  Tcl EPI returns its data through a number of global variables. If you are using these variables from within Tcl sub-routines, you need to declare them as having global scope with the global keyword.

  Example

  ```
  proc myProc {myArg} {
     global NMSEPI_RESULT;
          ...
     if {$NMSEPI_RESULT == 0}}
          ...
  ```

- **script headers**
  Since the Magellan Contrib package (which provides a version of the Tcl code) does not install the code in a typical place (*/opt/MagellanContrib*), scripts that use Tcl EPI should be written using a UNIX `#!` script header as follows:

```
#!/bin/sh
# The next line restarts using wish. \
exec wish "$0" "$@"
```

(Substitute `wish` for the Tcl-based Shell to use (for example, `expect, scotty`). As well, note the \ at the end of the comment line). Scripts with this header will use the Tcl shell that is available from the `PATH/path` environment. The Preside Multiservice Data Manager (MDM) user environment is modified to support the proper paths for this to work.

Base

In general, the first task in writing Tcl EPI scripts is to initialize an interface. There are specific API and Command Access commands to do this, but the Base also provides a general routine:

- **NMSEPIInit**
  Initializes the EPI environment.

- **NMSEPITerm**
  Drops all current API and command interfaces and terminates the IPC environment. This is sometimes required when another EPI script (which makes and maintains its own API or command interfaces) is reused and invoked in the same UNIX process. The second script would otherwise fail because of duplicate server connections.

EPI normally outputs error messages (for example, if the connection to a server fails to be created) on the standard error stream. You can prevent this from occurring by using the following command:

- **NMSEPISet +err|-err**
  Controls whether error messages are output (`+err`) or not output (`-err`).

API and Command Access commands usually apply to a default connection (a single connection for all APIs, another one for Command Access, and another one for Customer Database access). When using multiple concurrent connections, you can use the following commands to make a specific command the default and thereby avoid having to specify its name as an argument in every command:

- **NMSEPIDefaultAPI [-iname <name>]**
  NMSEPIDefaultCmd [-iname <name>]
  NMSEPIDefaultCdb [-iname <name>]

NMSEPIDefaultRTAC [-iname <name>]
NMSEPIDefaultNRS [-iname <name>]
If it exists, makes the named interface connection the default. If no interface is named, the original unnamed default interface, if any, is set to be the default. The last two calls are available only with the EPI reporting extension (RTAC and NRS interfaces).

Note: There is only one default API interface for all API types (for example, Alarm and Status, Network Model).

The following Preside Multiservice Data Manager (MDM) utilities are supported:

- **NMSEPIConvertCompId [-out] [-canon|-disp|-type|-mnem|-ep1|-ep2|-dpn|-switch] <component ID>**
  Converts the specified component ID and returns the result in the `NMSEPI_COMP_ID` global variable and as a string result (if `-out` is used). The conversions are as follows:

  `-canon`
  converts to canonical API format (for example, `PM AM1 PE 1 PI 1`).

  `-disp`
  converts to display format (for example, `PM/AM1 PE/1 PI/1`).

  `-type`
  extracts the module/link type (for example, `PM` or `NL`).

  `-mnem`
  extracts the module mnemonic (for example, `AM1`).

  `-ep1` and **`-ep2`**
  extract the first and second link endpoints in canonical API format.

  `-dpn`
  converts a DPN-100 OA or a PE/PI/PO component ID to a form suitable for commands (`<mnemonic> [pe <pe#>|<pi #> [<po #>]]`). (For example, `AM1 11`.).

-switch
returns the module-level component ID in canonical API format (for example, PM AM1).

-omni
returns the Preside Multiservice Data Manager (MDM) HP-OpenView DeskTop compatible component ID (similar to display format except for link names).

Examples

```
NMSEPIConvertCompId -ep1 "$myLinkId"
echo "Endpoint1: $NMSEPI_COMP_ID"

NMSEPIConvertCompId -dpn "$myDPNPort"
NMSCmdSendCommand $myOA "$NMSEPI_COMP_ID enable"
```

- **NMSEPICompareCompIds <component ID1> <component ID2>**
  Compares the two returning component IDs and returns (instead of the usual return codes) 0 if they are identical, <0 if the first component ID precedes the second, and >0 if the second precedes the first.

- **NMSEPIConvertTime [-out] [-stc] -api|-tostc|-epoch|-unix|-alarm|-pp |-ftime <format>|-offset <nb> [days] [<time string>]**
  Converts the specified time string and returns the result in the NMSEPI_OUTPUT_TIME global variable and as a string result (if -out is used). The input time can be in:

  API format (YYYY MM DD HH MM SS)
  Common Alarm format (YY-MM-DD HH:MM:SS)
  UNIX Epoch (<number of seconds since 1970>)
  Passport reply format (YYYY-MM-DD HH:MM_SS)

  The returned time is in the same time frame as the input one.However, exceptions occur if you specify -stc or if you select the -tostc conversion. If you specify -stc the input time is assumed to be in Standard Time Coordinates, that is, Greenwich Mean Time (GMT). If -stc is used and not -tostc then the output time is converted from STC to local workstation time. If the input time is not specified, the current workstation's time is used (and -stc is ignored).

The conversions are as follows:

`-api`
produces the time in API format.

`-tostc`
produces the time in API format converted to Standard Time Coordinates.

`-epoch`
returns the time as a UNIX Epoch value (number of seconds since 1970).

`-unix`
returns the time as the default time format for the workstation's LOCALE (see `man -s3c ctime`)

`-alarm`
returns the time in Common Alarm format.

`-pp`
reeturns the time in Passport reply format

`-ftime <format>`
returns the time in the specified format (see `man -s3c strftime`, 100 characters maximum).

`-offset <nb> [days]`
if `days` are not specified, returns the time in API format after applying the specified positive or negative offset in seconds, otherwise returns days

`-secsinday`
returns the number of seconds from the previous midnight and the specified time (or the current time)

`-stcoffset`
returns the number of seconds between the local workstation time and the coordinated universal time (UTC) and, if needed, taking daylight savings time into consideration. The offset is positive going west from UTC (same as UNIX timezone/altzone).

Example

```
NMSEPIConvertTime -offset -7 days
puts "$NMSEPI_OUTPUT_TIME\n"
```

returns the time 7 days ago, for example:

```
2000 03 24 17 01 19
```

```
NMSEPIConvertTime -epoch "2000 03 24 17 01 19"
puts "$NMSEPI_OUTPUT_TIME\n"
```

returns the same as UNIX epoch, for example:

```
953935279
```

```
NMSEPIConvertTime -ftime "Time was: %a %b %d %l:%M%p" \
                  "953935279"
puts "$NMSEPI_OUTPUT_TIME\n"
```

returns the same information using a custom format:

```
Time was: Fri Mar 24  5:01PM
```

- **NMSEPIPatternMatch [-out] [-g] <pattern> <target> [<substitute>]**

  Performs pattern matching of <pattern> with <target>. Specify <pattern> with grep syntax. You can include one \ (\) delimited sub-pattern, which is returned if there is a match. instead of the full matched portion. If you specify <substitute>, the matching sub-string is replaced by <substitute>. The `-g` option substitutes all matching sub-strings. The matched or substituted string is returned in the `NMSEPI_OUTPUT_MATCH` environment variable (and standard output if `-out` is used).

  Example

  ```
  if { [ NMSEPIPatternMatch "pe \(\[0-9\]*\)down" \
                "$NMSEPI_OUTPUT_TEXT" ] } {
     puts "PE# $NMSEPI_OUTPUT_MATCH is down.";
  ...
  ```

- **NMSEPIGetContext [-out] [-ssel|-user|-ws] <var. name>**
  NMSEPISetContext -user|-ws <var. name> <value>

  Gets or sets a Preside Multiservice Data Manager (MDM) context variable. The context corresponds to the Service Selection for `-ssel`, the current User Session for `-user`, and the Workstation Wide for `-ws`. `NMSEPIGetContext` sets the `NMSEPI_CTX_VALUE` global variable to the extracted value (and returned as a string if `-out` is used).

Example

```
NMSEPIGetContext -user "DPN_QUICK_STEP" ;
puts "Hot context: $NMSEPI_CTX_VALUE" ;
```

- **NMSEPILongArith [-out] [-u] <integer value> +|-|x|/|% <integer value>**

  Performs long integer arithmetic on the values provided (+ addition, - subtraction, x or * multiplication, / division, and % modulo). If -u is used, unsigned arithmetic is performed. The results are returned as a string in the NMSEPI_RESULT global variable and as a string result if -out is specified.

  Example

```
NMSEPILongArith $currByteCount - $prevByteCount
echo "Delta: $NMSEPI_RESULT"
```

The following commands are supported only in Tcl EPI, but do not function when the graphical Tk extension is used (explicit Recv routines must be used instead).

- **NMSEPIEventLoop**

  Initiates an Xt-based event loop for asynchronous message handling. This command never returns. The processing scripts are then performed by the callback Tcl code that is bound to the API and command interfaces (see NMSEPIRegisterContextInterest, NMSAPIBindCallback, NMSCmdBindCallback, and NMSCdbBindCallback).

- **NMSEPIRegisterContextInterest [-user|-ws <callback> <var. names...>]**

  Binds the specified Tcl command string, <callback>, (typically a function invocation and its arguments) to the specified Preside Multiservice Data Manager (MDM) Context variables and domain (-user for the User Session and -ws for Workstation Wide). This command string will be executed whenever one of the named variable changes value and the script is in an event loop (NMSEPIEventLoop). The following global variables are available to the callback:

  NMSEPI_CB_REASON

  is set to CONTEXT.

NMSEPI_CTX_VAR
is set to the the named of the changed variable.

NMSEPI_CTX_VALUE
is set to the new value for the variable.

Only one such callback can be registered. To deregister the callback, call this command without any arguments.

Example

```
proc componentHotContext {} {
    # react to the hot context selection
    # whose component name value is in
    # $NMSEPI_CTX_VALUE
    ...
}
...
NMSEPIRegisterContextInterest -user \
    "componentHotContext" "DPN_QUICK_STEP";
...
NMSEPIEventLoop;
```

- **NMSEPITimer set <msec> <callback>**
  | clear <timer name>

  The first form, `set`, of this command binds the specified Tcl command string, `<callback>`, (typically a function invocation and its arguments) to be invoked once in `<msec>` milli-seconds when the script is in an event loop (`NMSEPIEventLoop`). The created timer is given a name returned as the value of the `NMSEPI_TIMER_NAME` global variable. This name can be used in the second form, `clear`, to cancel the timer. The callback code is invoked with the following environment variables:

  NMSEPI_CB_REASON
  is set to `TIMER`.

  NMSEPI_TIMER_NAME
  is set to the name of the timer that has expired.

Example

```
proc rescanList {} {
    ...
    # re-create the timer to be called again
    NMSEPITimer set [ expr 5 * 60 * 1000 ] “rescanList”;
}
NMSEPITimer set [ expr 5 * 60 * 1000 ] “rescanList”;
...
NMSEPIEventLoop;
```

Generic API access

These commands provide access to the Generic API at the same level as if you were using the Generic API Provider utility, *genapi*, directly. At this level, no special knowledge of the individual API types is required; all queries can be handled in a generic way. (For more information, see 241-6001-200 *Preside MDM Application Programming Interface Primer*.

Generic API Access commands are used in the following sequence:

1 Initialize an API interface.

2 Connect to the API server.

3 If the server requires it, send a `REGISTER` message and wait for its reply.

4 Send an API query.

5 Receive the next reply and extract the needed information.

6 Disconnect from the API server.

7 Drop the API interface.

Multiple queries can be issued, but individual API interfaces will handle these queries synchronously, one at a time (except for sieve event notifications). It is possible, however, to create multiple-named API interfaces (each one resulting in its own connection to the server and therefore capable of independent, parallel queries). Note also that a single API interface can be reused (disconnected and reconnected to another server).

The same command sequence is used by the Specific API Access areas, but specialized commands are sometimes added to handle the particular details of an API type. For more information on specialized commands, see "Specialized API access" (page 156).

Generic API Access commands

The following Generic API Access commands are provided:

- **NMSAPIInit [-iname <name>] <API dictionary path>**
  Initializes a Generic API interface using the indicated API dictionary. If more than one interface is to be used, you should provide a name for it. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

  Note: Specialized API Access provides its own version of this command, which hides the detail of the API dictionary path. For more information, see "Specialized API access" (page 156).

- **NMSAPIDrop -iname <name>**
  Drops the named API interface to clear up the connection or so it can be reused.

- **NMSAPIConnect [-iname <name>] <service name> [<host>]**
  Connects the default or named interface to the specified API server, optionally on a different Preside Multiservice Data Manager (MDM) host.

  Example

  ```
  NMSAPIInit /opt/MagellanNMS/lib/api/GMDR.dict;
  if { [ NMSAPIConnect GMDR localhost ] } {
      ... # connection successful
  ```

- **NMSAPIDisconnect [-iname <name>]**
  Disconnects the default or named interface from its current API server.

- **NMSAPIRegister [-iname <name>] <user id> [<password>] [-attr <attribute lines>]**

  Sends a `REGISTER` message through the default or named interface with the specified parameters

  where:

  `-attr <API attribute lines>`
  is one or more API attribute lines to add to the message (for example, the `"_attr: userCapability E mdInject"` line needed to register to IMDR with alarm injection capabilities).

  This command is actually a combination call, since it will both send the query and wait for the reply. The return code therefore indicates the success or failure of the complete `REGISTER` command-reply sequence.

  Example

```
if { [ NMSAPIRegister toto \
     -attr "_attr: userCapability E mdInject" ] } {
... # successful register for alarm injection
```

- **NMSAPISendCommand [-iname <name>] <API query string>**

  Sends a query to the default or named interface. The query is specified as an ASCII string in API syntax.

  Example

```
NMSAPISendCommand "_cmd: get
_obj_type: network
_obj_id: networkID S compRoot
_scope: all
_attr_id: all
_filter:compID LEFT NI PM"
```

- **NMSAPIRecvReply [-iname <name>] [-out] [-var <array name>] [<timeout>]**

  Waits for and receives the next reply record from the server. A timeout

can be specified (by default, it waits forever). With a timeout of `0`, the command acts as a no-wait poll. If `-out` is specified, the received record is returned as a string in an API-like format instead of the exit code.

If a variable name is specified with `-var`, the named variable is typeset as an associative array containing the received API fields. For API fields having an attribute name, the array key is set to:

```
<API label (without the :)>,<API attribute name>
```

and for the others, the key is set to:

```
<API label (without the :)>
```

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

Example

Receving a GET command reply for a Network Model node could lead to the following values:

```
while { [ NMSAPIRecvReply -var reply ] } {
   #...
   # $reply(_obj_type) is "node"
   # $reply(_obj_id,compId) is "PM R78"
   # $reply(_attr,rawState) is "INSV"
   #...and so on...
```

It is possible to list all the array keys using Tcl's "`array names <array name>`" construct.

The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred).

- **NMSAPISkipRestOfReply [-iname <name>]**
  This combination call waits for and ignores all further replies until the end of response. If it finds an error record, the `NMSEPI_RECORD_TYPE` global variable is set to `ERROR`; otherwise it is set to `ENDRESP`.
- **NMSAPIGetRecord [-iname <name>] [-out] [-var <array name>]**
  Extracts the last received record's (`NMSAPIRecvReply`) record type (`NONE`, `REGISTER`, `RESPONSE`, `EVENT`, `ENDRESP`, `ERROR`, or `END`). It places this information in the `NMSEPI_RECORD_TYPE` global variable (and returned it as a string if `-out` is used) and resets the API field list

to the beginning for `NMSAPIGetNextField`, `NMSAPIFindNextField`, and `NMSAPIFindNextAttr` (it can therefore be called several times to repeatedly scan the field/attribute list).

If a variable name is specified with `-var`, the named variable is typeset as an associative array containing the received API fields. For API fields having an attribute name, the array key is set to:

```
<API label (without the :)>,<API attribute name>
```

and for the others, the key is set to:

```
<API label (without the :)>
```

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

It is possible to list all the array keys using Tcl's "`array names <array name>`" construct.

Example

```
while { [ NMSAPIRecvReply ] } {
                NMSAPIGetRecord
   if { "$NMSEPI_RECORD_TYPE" = "ENDRESP" } {
                          ... # got end of response
```

- **NMSAPIGetNextField [-iname <name>] [-out]**
  Extracts the next API field from the last received reply. The following global variables are set:

  `NMSEPI_FIELD_LABEL`
  is the API field type (`_attr:`, `_end_resp:`, `_error:`, `_end:`, `_obj_type:`, `_obj_id:`, `_user_id:`, `_capability:`, `_inv_id:`, `_event_type:`, `_time:`, or `_sieve_id:`).

  `NMSEPI_FIELD_NAME`
  is the API field name element (`_ATTR:` and `_OBJ_ID:` only).

  `NMSEPI_FIELD_TYPE`
  is the API field type element (`_ATTR` and `_OBJ_ID` only).

  `NMSEPI_FIELD_VALUE`
  is the API field value.

Multiple-line attribute block values are returned as a single multiple-line value.

If `-out` is used, the field is returned as a string in an API-like format.

Example

```
NMSAPIGetRecord;
...
while { [ NMSAPIGetNextField ] } {
   if {   "$NMSEPI_FIELD_LABEL" == "_attr:"
       && "$NMSEPI_FIELD_NAME"  == "time" } {
      puts "Time is: $NMSEPI_FIELD_VALUE";
...
```

- **NMSAPIFindNextField [-iname <name>] [-out] <label>**
  Locates and returns the next API line of the type specified by `<label>` (`_attr:`, `_end_resp:`, `_error:`, `_end:`, `_obj_type:`, `_obj_id:`, `_user_id:`, `_capability:`, `_inv_id:`, `_event_type:`, `_time:`, or `_sieve_id:`) from the API field list of the last received API reply. The global variables are set as for `NMSAPIFindNextField`. If `-out` is used, the field is returned as a string in an API-like format (but without the label) on standard output. The previous example can be rewritten as follows:

  ```
  NMSAPIGetRecord;
  ...
  while { [ NMSAPIFindNextField "_attr:" ] } {
     if { "$NMSEPI_FIELD_NAME" = "time" } {
        puts "Time is: $NMSEPI_FIELD_VALUE";
  ...
  ```

- **NMSAPIFindNextAttr [-iname <name>] [-out] <attr. name>**
  Locates and returns the next named attribute from the API field list of the last received API reply. The global variables are set as for `NMSAPIFindNextField`. If `-out` is used, the field value is returned as a string. The previous example can be rewritten as follows:

  ```
  NMSAPIGetRecord;
  ...
  if { [ NMSAPIFindNextAttr "time" ] } {
     puts "Time is: $NMSEPI_FIELD_VALUE";
  ...
  ```

The following command is supported only in Tcl EPI. It does not function when the graphical Tk extension is used (explicit `Recv` routines must be used instead).

- **NMSAPIBindCallback [-iname <name>] <callback command>**
  Binds the specified Tcl command string (typically a function invocation and its arguments) to the API interface. This command string will be executed whenever a new message is received from the server for this interface and the script is in an event loop (`NMSEPIEventLoop`). The following global variables are available to the callback:

  `NMSEPI_INAME`
  is the API interface name of the callback.

  `NMSEPI_CB_REASON`
  is the reason for the callback (`LOST_CONNECTION`, `ERROR`, `EVENT`, or `RESPONSE`).

  `NMSEPI_RECORD_TYPE`
  is the returned record (API record type or `ALARM` if the received record is an Alarm, or `RAWSTATE` if it is a Raw State).

  `NMSEPI_ALARM_SEVERITY, NMSEPI_ALARM_EVENT, NMSEPI_ALARM_FAULT, NMSEPI_ALARM_TIME, NMSEPI_ALARM_COMPID, NMSEPI_ALARM_OPDATA`
  are the special alarm fields

  `NMSEPI_ALARM_TIME, NMSEPI_ALARM_COMPID, NMSEPI_RAW_STATE`
  are the alarm fields if a Raw State Change record is received

  `NMSEPI_API_SIEVEID`
  is the source Sieve ID for the received message, if any.

  In addition, the post-`Recv` functions (i.e., `NMSAPIGetRecord`, `NMSAPIGetNextField`) can be used in the context of this callback to extract other API fields and attributes from the received reply. The event loop is launched with `NMSEPIEventLoop` and never returns.

Example

```
proc mycb {} {
   # ...Process the reply...
}
# ... launch commands and create sieves ...
NMSEPIBindCallback mycb;
...
NMSEPIEventLoop;  # never returns
```

Specialized API access

This section describes additional commands as well as value-added versions of the Generic API Access commands that are built to interact with specific APIs and their features.

Alarm and Status API Access

This area adds commands to interact with the Alarm and Status API. In addition to a number of combination calls (for example, to create an alarm sieve or to inject alarms), it also supports automatic extraction of the major alarm attributes to support most forms of network automation.

Alarm and Status API Access commands

The following Alarm and Status API Access commands are provided:

- **NMSGMDRAPIInit [-iname <name>]**
  This simplified form of the `NMSEPIInit` command initializes an API interface to the Alarm and Status API server (GMDR). If multiple API interfaces are to be used, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

- **NMSGMDRAPIConnect [-iname <name>] [-ssel | <host>]**
  Connects to the Service Selected `GMDR` server (if `-ssel` is specified), the specified location (`<host>`), or the local host (by default).

  Example

  ```
  NMSGMDRAPIInit -iname toto
  if { [ NMSGMDRAPIConnect -iname toto -ssel ] } {
      ... # we're connected
  ```

- **NMSGMDRAPICreateAlarmSieve [-iname <name>] [-all] [-attr <alarm event filters and attributes...>]**

  This combination call creates a simplified alarm sieve and waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the major alarm attributes (`compId`, `time`, `severity`, `event`, `faultCode`, and `operatorData`) of all of the received alarms. However, if `-all` is specified, all of the attributes are extracted. You can add event filter (`"_attr: eventFilter SS <attribute> <operator> <type> <value>"`) and specific attribute extraction (`"_attr: eventInfo S <attribute>"`) parameters in API syntax using the `-attr` option and its values (note that each value may be multiple-lined).

  Example

  ```
  if { [ NMSGMDRAPICreateAlarmSieve -attr \
      "_attr: eventFilter SS event EQ E SET" \
      "_attr: eventInfo S commentData" ] } {
      ... # sieve creation successful
  ```

  The notifications from the sieve(s) can be collected with `NMSAPIRecvReply` or with `NMSGMDRAPIRecvAlarm` (with the EVENT record type).

  Each Alarm and Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned as the `NMSEPI_API_SIEVEID` variable.

  Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **NMSGMDRAPIRecvAlarm [-iname <name>] [-out] [-var <array name>] [<timeout>]**

  Enhances `NMSAPIRecvReply` by waiting for and receiving the next reply and automatically extracting the major alarm fields in their own global variables (if the reply is the result of an alarm *get* or notification). The following global variables are set:

NMSEPI_RECORD_TYPE:
is the API record type.

NMSEPI_ALARM_SEVERITY:
is the severity of the alarm.

NMSEPI_ALARM_EVENT
is the alarm event.

NMSEPI_ALARM_FAULT
is the fault code.

NMSEPI_ALARM_TIME
is the alarm time.

NMSEPI_ALARM_COMPID
is the component ID.

NMSEPI_ALARM_OPDATA
is the operator data.

For information on the legal values of these fields, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The other attributes, if any were specified, can be extracted with the `NMSAPIGetRecord`, `NMSAPIGetNextField`, `NMSAPIFindNextField`, and `NMSAPIFindNextAttr` commands.

Example

```
NMSGMDRAPICreateAlarmSieve -attr \
   "_attr: eventFilter SS event EQ E SET";
...
if { [ NMSGMDRAPIRecvAlarm ] } {
```

```
if {   "$NMSEPI_RECORD_TYPE"    == "EVENT"
    && "$NMSEPI_ALARM_SEVERITY" == "critical" } {
   ... # handle the critical alarm
```

If `-out` is used, the default alarm fields are returned as a string consisting of a sequence of at least six lines (for the `severity, event, faultCode, time, compId,` and `operatorData` respectively).

If a variable name is specified with `-var`, the named variable is typeset as an associative array containing the received API fields. For API fields having an attribute name, the array key is set to:

`<API label (without the :)>,<API attribute name>`

and for the others, the key is set to:

`<API label (without the :)>`

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

Note: Some lines may be empty if the corresponding field is not present. As well, `operatorData` can consist of more than one line.

- **NMSGMDRAPIFormatAlarm [-iname <name>] [-out] [terse|normal|full]**

  This command produces the last received alarm (from `NMSAPIRecvReply`, `NMSGMDRAPIRecvAlarm` or from a callback) in the Preside Multiservice Data Manager (MDM) Common Alarm Format (as used in the Alarm Display and Component Information Viewer tools). The alarm can be formatted in either `terse` (one line), `normal` (includes Operator Data) or `full` (all information) formats, the default is `full`. The `NMSEPI_ALARM_DISPLAY` variable is set to the resulting string.

  If `-out` is used, the resulting string is also returned.

- **NMSGMDRAPIInjectAlarm [-iname <name>] <comp ID> <event> <severity> <fault code> <notification ID> <comment> [-time <time>] [-attr <other attributes...>]**

  Sends an alarm injection command with the following specified parameters:

`comp ID`
is the component ID.

`event`
is the alarm event (`set|clear|message`).

`severity`
is the severity of the alarm (`warning|minor|major|critical|cleared|indeterminate`).

`fault code`
is the fault code in `AAAACCCC` format.

`notification ID`
is the sequence number (if 0 or an empty string, a unique value will be provided).

`comment`
is the comment data string.

As well, a time can be specified using `-time` (in *`yyyy mm dd hh mm ss`* format) and other attributes specified using `-attr` and its values (as one or more `"_attr: <name> <type> <value>"` per string). Unspecified attributes take default values (see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*).

Note: This is not a combination call since the command does not wait for the server's reply, which must then be explicitly received or ignored.

Example

```
NMSGMDRAPIInjectAlarm "MYRTR WEST3" \
      major set A0000001 23 \
      "The router does not reply." \
      -attr "_block: _attr operatorData S
Try: 12, Timeout: 5, IP: 33.24.1.1
_end_block:"
NMSAPISkipRestOfReply
```

- **NMSGMDRAPICreateRawStateSieve [-iname <name>] [-attr <event filters...>]**

  This combination call creates a simplified raw state change sieve and

waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the `compId`, `time`, and `rawState` attributes. It is possible to add an event filter (`"_attr: eventFilter SS <attribute> <operator> <type> <value>"`) using the `-attr` option and its values (note that each value may be multi-lined).

Example

```
if { [ NMSGMDRAPICreateRawStateSieve -attr \
    "_attr: eventFilter SS rawState EQ E OOS" ] } {
    ... # sieve creation successful
```

The notifications from the sieve(s) can be collected with `NMSAPIRecvReply` or with `NMSGMDRAPIRecvRawState` (using the EVENT record type).

Each Alarm and Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned as the `NMSEPI_API_SIEVEID` variable.

Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **NMSGMDRAPIRecvRawState [-iname <name>] [-out] [-var <array name>] [<timeout>]**

  Enhances `NMSAPIRecvReply` by waiting for and receiving the next reply and automatically extracting the raw state change fields in their own global variables (if the reply is the result of a node *get* or notification). The following global variables are set:

  `NMSEPI_RECORD_TYPE`
  is the API record type.

  `NMSEPI_ALARM_TIME`
  is the notification time.

NMSEPI_ALARM_COMPID
is its component ID.

NMSEPI_RAW_STATE
is its raw state value.

For information on the legal values of these fields, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*.

Example

```
NMSGMDRAPICreateRawStateSieve -attr \
                          "_attr: eventFilter SS
rawState EQ E OOS";
...
if { [ NMSGMDRAPIRecvRawState ] } {
   if { "$NMSEPI_RECORD_TYPE" == "EVENT" } {
      ... # handle the out-of-service component
```

If `-out` is used, the raw state fields are returned as a string consisting of a sequence of three lines (for the `rawState, time,` and `compId` respectively).

If a variable name is specified with `-var`, the named variable is typeset as an associative array containing the received API fields. For API fields having an attribute name, the array key is set to:

```
<API label (without the :)>,<API attribute name>
```

and for the others, the key is set to:

```
<API label (without the :)>
```

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

Network Model API access

The Network Model (NM) API access allows you to initialize an API interface to the server and connect the interface to the host.

Network Model API Access commands

Network Model (NM) API Access provides two additional commands:

- **NMSNMAPIInit [-iname <name>]**
  Initializes an API interface to the NM Server. If multiple API interfaces are to be used, specify a name in the command invocations for this

interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

- **NMSNMAPIConnect [-iname <name>] [-ssel | <host>]**
  Connects the interface to the Service Selected NM Server host (`-ssel`), the named host, or the local host (by default).

  Example
  ```
  NMSNMAPIInit
  if { [ NMSNMAPIConnect bcars561 ] } {
      ... # we're connected
  ```

Various Generic API Access commands can be used to send queries and receive their replies.

Host Group Directory Service API access

The Host Group Directory Service (HGDS) API Access allows you to extract data on the Passport Group configuration of Preside Multiservice Data Manager (MDM). Some value-added and combination calls are added to the Generic API Provider along with automatic extraction of the Passport Member information.

HGDS API Access commands

The following HGDS API Access commands are provided:

- **NMSHGDSAPIInit [-iname <name>]**
  Initializes an API interface to the Host Group Directory Server. If multiple API interfaces are to be used, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

- **NMSHGDSAPIConnect [-iname <name>] [-ssel|<host>**
  Connects the interface to the Service Selected HGDS (`-ssel`), the named host, or the local host (by default).

  Example
  ```
  NMSHGDSAPIInit -iname hgds;
  if { [ NMSHGDSAPIConnect -iname hgds -ssel ] } {
     ... # we're connected
  ```

- **NMSHGDSAPISendQuery [-iname <name>]**
  -group [<group name>]
  | -member [<member name>]
  | -child <group name>
  | -parent <member name>

  Sends an HGDS API query. Use `-group` to retrieve the named Passport Group (or all of them if no name is indicated), `-member` for the named Passport module (or all of them if no name is indicated), `-child` for the Passport hosts in the named group, or `-parent` for the groups containing the named host.

- **NMSHGDSAPIRecvReply [-iname <name>] [-out] [<timeout>]**
  Enhances `NMSAPIRecvReply` to automatically extract the Passport host information if present in the global variables. The following global variables are set:

  NMSEPI_RECORD_TYPE
  is the reply record type.

  NMSEPI_HGDS_NAME
  is the Passport host or group name.

  NMSEPI_HGDS_IPADDR
  is the Passport host IP address, if applicable.

  Example:

```
if { [ NMSHGDSAPISendQuery -iname hgds \
             -child "$grp" ] } {
  while { [ NMSHGDSAPIRecvReply -iname hgds ] } {
      exec ping $NMSEPI_HGDS_IPADDR;
      if { errorCode == 0 } {
         ... # Passport is reachable
```

Command access

Like the `cmccmd` utility, Command Access allows scripts to connect to Passport Groups and DPN-100 OAs (the command route), send commands to the modules they contain, and receive the replies. Command Access also provides several utility commands to assist with the parsing and identification of the command output. Command Access commands further communicate with the Command Session servers (CMCFUN and CM) that correspond to the current `DISPLAY` environment variable (for example, for a script that uses

the session servers in the current Preside Multiservice Data Manager (MDM) User Session and potentially uses its current group and OA connections) in the Command Console.

For stand-alone (for example, CRON) or specific macros (for example, using Passport provisioning mode), it may be necessary to create a private Command Session for the execution of the script. Command Access provides the `NMSCmdSession` comand to start (and stop) a session. (For a description of `cmcwrap` and how to use it to write macros, see 241-6001-301 *Preside MDM Customization Administrator Guide*).

Command Access commands are used in the following sequence:

1 Initialize a command interface.

2 Connect to the Command Session server (CMCFUN).

3 Connect to one or more Passport Group(s) or OA(s), if necessary.

4 Send a command to a node in the connected Groups and OAs.

5 Receive the command replies, either as a single string or one line at a time.

6 Use Tcl or the provided commands to analyze the reply.

7 Disconnect from the command server.

8 Drop the command interface.

Multiple commands can be sent to the nodes, but individual command interfaces will handle these commands synchronously, one at a time. It is possible, however, to create multiple named command interfaces (each one resulting in its own connection to the server and therefore capable of independent, parallel commands).

Command Access commands

The following Command Access commands are provided:

- **NMSCmdSession [-display <DISPLAY>] start|stop**
  [-dpn <MDM host>] [-pp <MDM host>]
  [-idle <timeout>]

  Starts (`start`) or stops (`stop`) a private command session for the

specified <DISPLAY> name (must be a unique value workstation-wide for private sessions). This is equivalent to spawning the following command in the background:

```
/bin/env DISPLAY=<DISPLAY> \
   /opt/MagellanNMS/bin/loop
          -delay 3 \
               /opt/MagellanNMS/bin/icm \; \
               /opt/MagellanNMS/bin/cmcfun
```

Since loop is used, the session terminates automatically within 30 seconds if the script that has spawned it terminates without stopping it (be careful of middle shell processes). See 241-6001-301 *Preside MDM Customization Administrator Guide* for more information on the loop utility.

The calling script waits for two to three seconds after invoking this command (or performs whatever operations it wants to) before attempting to connect to the session to let the session servers initialize themselves properly.

If one of -dpn and/or -pp is specified, the matching service selection is applied to the new session so that the corresponding servers are used instead of the current workstation service selection. See NMSCmdSetServiceSelection to change the session's service selection.

If -idle is specified with a non-zero value, the time (in minutes) is passed to cmcfun which self-terminates automatically if the specified time elapses with no command activity. This ensures that device connection resources are freed if not used for an extended period of time. The next time the session is used, the required group connection needs to be re-created.

Example

```
# start a private command session and wait for it
# to initialize
NMSCmdSession -display mysession.$$ \
|| exit 1
```

```
sleep 3
# initialize an connect the command interface
# with -display mysession.$$ (see below)
```

- **NMSCmdInit [-iname <name>]**
  Initializes a command interface. If multiple interfaces are to be used at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultCmd` to make it the default interface).

- **NMSCmdDrop -iname <name>**
  Drops the named command interface (frees allocated resources).

- **NMSCmdConnect [-iname <name>] [-display <DISPLAY>]**
  Connects the interface to the current Command Session servers (for interactive scripts used within an Preside Multiservice Data Manager (MDM) user session or as argument to `cmcwrap`) or the servers that correspond to the specified `DISPLAY` variable for a previously started alternate Command Session server set.

  Example

```
NMSCmdInit;
if { [ NMSCmdConnect -display $mysession ] } {
... # we're connected
```

- **NMSCmdDisconnect [-iname <name>]**
  Disconnects the interface from the session servers. The interface may be reconnected to the servers from the same session or to servers from another session.

- **NMSCmdSetServiceSelection [-iname <name>]**
  [-dpn <MDM hostname>]
  [-pp <MDM hostname>]
  Controls the Service Selection settings for the Command Session this interface is connected to. `-dpn` specifies the name of the MDM host to be used for DPN Network Access. `-pp` specifies the name of the MDM host to be used for Passport Network Access.
  This is similar to using the Service Selection tool on the same session as the Session Servers so if other users (or programs/scripts) are using that

session, then they will also be impacted by this change. The Command Interface must be connected, see `NMSCmdConnect`) for this function to work.

- **NMSCmdSendConnect [-iname <name>] OA|GROUP <route> <name> <password>**

  Sends a connect request to the indicated route using the specified authentication information. If an error is found, the error text is the value of the `NMSEPI_OUTPUT_TEXT` global variable.

  Note: Connecting to an already connected route results in an error, even though the route is available. Send a `""` command to the route to verify if a connection has already been established. If the connection does not exist, the reply indicates an error.

  Example

```
NMSCmdSendCommand myOA ""
if {   [ NMSCmdSkipRestOfReply ]
    || [ NMSCmdSendConnect OA \
               myOA myCap aq1sw2 ] } {
   ... # we're connected to the route
```

  Note: If the script is to be invoked from the Preside Multiservice Data Manager (MDM) Command Console, you can use `"$ENV{'CMC_CURRENT_ROUTE'}"` (with the quotes as indicated) here and in `NMSCmdSendCommand` to indicate the current Command Console route.

- **NMSCmdSendDisconnect [-iname <name>] <route>**

  Disconnects the interface from the named route.

- **NMSCmdSendCommand [-iname <name>] <route> <command>**

  Sends a command to a node through the specified connected route. The name of the destination node is typically the first token of the command.

  Example

```
if { [ NMSCmdSendCommand myOA "R78 d" ] } {
   ... # command sent
```

Note: The special command routes of the Command Console ($ for macro access, @ for the SNMP Command Framework, and * for the Passport wild-card group) can all be used as routes from EPI though, obviously, there is no need to first connect to them.

Note: Be careful when specifying, in the node command, special characters (for example, `*` or `?`) which may be interpreted and substituted by Tcl. You should quote the command string as a whole.

- **NMSCmdRecvFullReply [-iname <name>] [-out]**
  Waits for and receives the complete reply to the previous node command. The replied text is available in the `NMSEPI_OUTPUT_TEXT` global variable (and returned as a string if `-out` is used). The received text is also available to the `NMSCmdPatternMatch` command.

  Example
  ```
  NMSCmdSendCommand myOA “dir”;
  if { [ NMSCmdRecvFullReply ] } {
     exec echo “$NMSEPI_OUTPUT_TEXT” \
          | grep “pe”;
  ...
  ```

- **NMSCmdRecvNextLine [-iname <name>] [-out] [<timeout>]**
  Waits for and receives the next line of the last reply of the issued node command. A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts a no-wait poll. The replied text is available in the `NMSEPI_OUTPUT_TEXT` global variable (and returned as a string if `-out` is used). The received text is also available to the `NMSCmdPatternMatch`, `NMSCmdGetNumColumns`, and `NMSCmdGetColumn` commands. The previous example can be rewritten as follows:

  ```
  NMSCmdSendCommand myOA “r78 d”;
  while { [ NMSCmdRecvNextLine ] } {
     if { [ regexp “.*pe.*” \
                  “$NMSEPI_OUTPUT_TEXT” ] } {
        puts “$NMSEPI_OUTPUT_TEXT”
  ...
  ```

- **NMSCmdRecvNextChunk [-iname <name>] [-out] [<timeout>]**
  Waits for and receives the next chunk of the last reply of the issued node command. A chunk is most efficiently processed by EPI and may contain

multiple lines of text (it can even end in the middle of a line). A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts a no-wait poll. The replied text is available in the `NMSEPI_OUTPUT_TEXT` global variable (and returned as a string if `-out` is used). The received text is also available to the `NMSCmdPatternMatch`, `NMSCmdGetNumColumns`, and `NMSCmdGetColumn` commands. The previous example can be rewritten as follows:

```
NMSCmdSendCommand myOA "r78 d";
while { [ NMSCmdRecvNextChunk ] } {
   if { [ regexp ".*pe.*" \
               "$NMSEPI_OUTPUT_TEXT" ] } {
      puts "$NMSEPI_OUTPUT_TEXT"
...
```

- **NMSCmdSkipRestOfReply [-iname <name>]**
  This combination call waits for and ignores all replies until the end of response.

- **NMSCmdGetNumColumns [-iname <name>]**
  Returns the number of blank (space or tab) separated columns in the previously received reply line. The column count is returned by the routine instead of an error code as well as in the `NMSEPI_NUM_COLUMNS` global variable.

- **NMSCmdGetColumn [-iname <name>] [-out] <column> [...] [<target>]**
  Without `<target>`, returns the indicated blank (space or tab) separated column of the previously received reply line as the `NMSEPI_OUTPUT_COLUMN` global variable (and returned as a string if `-out` is used). If '`...`' is specified, all of the remaining columns starting at the given column are returned (blank separated). Column indexes start at `1`.

  Example

```
NMSCmdSendCommand myPassp "pp1 d fruni/101";
while { [ NMSCmdRecvNextLine ] } {
   NMSCmdGetColumn 1;
   att="$NMSEPI_OUTPUT_COLUMN";
   NMSCmdGetColumn 3 ...;
   val="$NMSEPI_OUTPUT_COLUMN";
```

```
   if { "$att" == "operationalState" } {
      puts "State: $val";
...
```

If <target> is specified, this command also checks whether the column has the same contents as the target and returns accordingly.

Example

```
NMSCmdSendCommand myPassp "pp1 d fruni/101";
while { [ NMSCmdRecvNextLine ] } {
   if { [ NMSCmdGetColumn 1 \
                "operationalState" ] } {
      puts "State: [NMSCmdGetColumn -out 3 ...]";
...
```

- **NMSCmdPatternMatch [-iname <name>] [-out] [-g] <pattern> [<substitute>]**

  This is the functionality of NMSEPIPatternMatch applied to the last received command response (full, line, or chunk).

  Example

```
NMSCmdSendCommand myOA "dir";
while { [ NMSCmdRecvNextLine ] } {
   if { [ NMSCmdPatternMatch ".*pe.*" ] } {
   puts "$NMSEPI_OUTPUT_TEXT";
...
```

- **NMSCmdSendDestRequest [-iname <name>][OA|GROUP|ALL]**

  This special utility command requests a list of all OA or Passport Group (or both) types of available routes. It corresponds to the cmccmd list command.

- **NMSCmdRecvNextDest [-iname <name>] [-out] [<timeout>]**

  Waits for and receives the next reply to an NMSCmdSendDestRequest command. The reply is returned with the following global variables:

  NMSEPI_OUTPUT_TEXT
  is the full reply text line.

  NMSEPI_DEST_NAME
  is the route name.

NMSEPI_DEST_TYPE
is the route type (OA or GROUP).

NMSEPI_DEST_STATE
is the route connection state (CONN, AUTH, or -).

Example

```
NMSCmdSendDestRequest GROUP;
while { [ NMSCmdRecvNextDest ] } {
   NMSCmdGetColumn 2;
   if { "$NMSEPI_OUTPUT_COLUMN" == "CONN" } {
      ... # route is available
```

The following command is supported only in Tcl EPI. It does not function when the graphical Tk extension is used (explicit Recv routines must be used instead).

- **NMSCmdBindCallback [-iname <name>] [-chunk|-ppcomp] <callback command>**
  Binds the specified Tcl command string (typically a function invocation and its arguments) to the command interface. Execute this command string whenever a new line (default), a new chunk (-chunk), or a new Passport component (-ppcomp) is specified or if a reply is received from the server for this interface. The following environment variables are available to the callback:

  NMSEPI_INAME
  is the callback command interface name.

  NMSEPI_CB_REASON
  is the reason for the callback (LOST_CONNECTION, ERROR, ENDRESP, or RESPONSE or FLOW CALLBACK).

  NMSEPI_OUTPUT_TEXT
  is the next line (default) or chunk (-chunk mode) of output from the command.

  NMSEPI_PPCOMPID
  is the the component ID of the next Passport component (-ppcomp

mode).

The text manipulation commands previously described (`NMSCmdGetNumColumns`, `NMSCmdGetColumn`, and `NMSCmdPatternMatch`) are also available in the context of the callback in line and chunk mode. In Passport component mode, use `NMSCmdGetPPCompID`, `NMSCmdResetPPCompAttrs`, `NMSCmdGetFirstPPCompAttr`, `NMSCmdGetNextPPCompAttr`, and `NMSCMDFindNextPPCompAttr` to extract the Passport component information. The event loop is launched with `NMSEPIEventLoop` and never returns.

Example

```
proc mycb {} {
   # ...Process the reply...
}
...
NMSCmdSendCommand myOA “r72 q serv”;
NMSCmdBindCallback mycb;
...
NMSEPIEventLoop;  # never returns
```

- **NMSCmdRecvNextPPComp [-iname <name>] [-out] [-var <var name>] [<timeout>]**

  `NMSCmdRecvNextPPComp` is is a form of `NMSCmdRecvFullReply` that waits for and receives the next Passport component reply to the previous command (i.e. the result of a `list` or `display` command). The name of the received component is available in the `NMSEPI_PPCOMPID` global variable (and returned as a sting if `-out` is used) and any error message is available in the `NMSEPI_OUTPUT_TEXT` global variable.

  Note: Make sure you always specify the `-notab` (no tabular output) CAS command line option when sending a Passport CAS display command with wildcards if this command is to be used to extract the replies.

  The `NMSCmdGetPPCompID`, `NMSCmdGetNextPPCompAttr`, and `NMSCmdFindNextPPCompAttr` commands can then be used to extract the replied component information. In addition, if a variable is specified with the `-var` option, the named variable is set as an associative array whose elements are the individual component attribute values. The

element key is the returned attribute name. If the attibute is a list, vector or array, the first index is added to the key with a coma as separator (for example, `$rep(pktFromIfByPrio,ep0)`). For two dimensional arrays, the entry is created with the attribute name as key and the column title as value. Another entry is created with "`<attribute name>,<row title>`" as key and the list of column labels as value. The remaining entries for this attribute have "`<attribute name>,<row label>`" as index and the list of corresponding columns as values. Finally, the entry key can also be one of the following special values; `Message`, contains any message emitted by Passport not part of of an attribute value (i.e. error messages), `CompID`, is the returned component's name.

It is possible to list all the array keys using Tcl's "`array names <array name>`" construct.

Example

```
NMSCmdSendCommand myGroup \
   "TOTO display shelf card/* utilization";
while { [ NMSCmdRecvNextPPComp -var reply ] } {
   puts "Card: $NMSEPI_PPCOMPID";
   puts "Average CPU: $reply(cpuUtilAvg)";
...
```

Other examples, using a display of the `Module-Vcs` and Passport `Shelf-Card` components, are:

```
# Simple attribute:
   $reply(cpuUtil) ... "5 %"
# SET value
   $reply(highPriorityPacketSizes) ..
              "16 32 64 128 256..."
# Vector values
   $reply(memoryUsage,fastRam) = "0 kbyte"
   #... and so on ...
# 2-D array values
   $reply(windowSize) ... "throughputClass"
   $reply(windowSize,packetSize) ...
              "0 1 2 3 4 ..."
   $reply(windowSize,16) ... "4 4 4 4 4 ..."
   #... and so on ...
# error
   $reply(Message) ... "   Component is disabled."
   $reply(localMsgBlockCapacity) ... "? kbyte"
```

- **NMSCmdGetPPCompID [-iname <name>] [-out] [-var <var name>]**
  `NMSCmdGetPPCompID` extracts the name of the last received Passport component in the `NMSEPI_PPCOMPID` global variable. If `-out` is specified, the name is also output on the standard output stream . If an associative array variable name is specified with `-var`, the array is also filled with the component's attributes as described for the `NMSCmdRecvNextPPComp` command. Finally, the list of Passport component attribute is reset to the beginning for the `NMSCmdGetNextPPCompAttr` and `NMSCmdFindNextPPCompAttr` commands.

  Example

  ```
  ...
  while { [ NMSCmdRecvNextPPComp -iname alt \
                -var reply ] } {
     NMSCmdGetPPCompID -iname alt;
     puts "Component: $NMSEPI_PPCOMPID";
  ...
  ```

- **NMSCmdGetNextPPCompAttr [-iname <name>] [-out]**
  `NMSCmdGetNextPPCompAttr` extracts the next attribute from the last received Passport component. The attribute name (and index, see the discussion on the associative array in `NMSCmdRecvNextPPComp`) is returned as the `NMSEPI_PPATTR_NAME` and its value as the `NMSEPI_PPATTR_VALUE` global variables. If `-out` is specified, both values are also returned as a single string with a space as separator.

  Example

  ```
  ...
  while { [ NMSCmdGetNextPPCompAttr ] } {
     puts "Attribute: $NMSEPI_PPATTR_NAME";
     puts "Value: $NMSEPI_PPATTR_VALUE";
  ...
  ```

- **NMSCmdFindNextPPCompAttr-iname <name>] [-out]**
  <attribute name>
  `NMSCmdFindNextPPCompAttr` extracts the next attribute from the last received Passport component whose name (in index, see the discussion on the associative array in `NMSCmdRecvNextPPComp`) matches the specified value. The attribute name/index is returned as the

NMSEPI_PPATTR_NAME and its value as the NMSEPI_PPATTR_VALUE global variables. If -out is specified, both values are also returned as a single string with a space as separator.

Example

```
...
if { [ NMSCmdFindNextPPCompAttr "bytesSent" ] } {
    NMSEPILongArith $prev - $NMSEPI_PPATTR_VALUE;
    puts "Delta: $NMSEPI_RESULT";
    set prev $NMSEPI_PPATTR_VALUE;
...
```

- **NMSCmdDoCommandFile [-iname <name>] [-out] [-trace[:[C][R][E][X]]] [-test] [-cb] [-bestEffort] [-avl <var name>] [<var>=<value> ...] <dest> <file path>**

  NMSCmdDoCommandFlow[-iname <name>] [-out] [-trace[:[C][R][E][X]]] [-test] [-cb] [-bestEffort] [-avl <var name>] [<var>=<value> ...] <dest> <command flow>

  These calls support the synchronous execution (commands and replies) of command flows from the file identified by <filePath> or the string identified by <commandFlow>. A command flow is a sequence of device commands (one per line) to be executed as one. The file may be specified as a formal path (starting with '/', '.' or '. .', or '~' which is automatically substituted to the user's HOME account), or as a relative path in which case the file is automatically searched for; first, in $HOME/MagellanNMS/<file path>, then /opt/MagellanNMS/cfg/<file path>, and finally /opt/MagellanNMS/lib/<file path>.

  All commands are executed through the named destination (if empty -- ""--, the destination must be prefixed to every device command in the flow). The execution of the flow terminates at the first failure unless -bestEffort is specified. Variable substitution on each command in

the flow is supported with the associative array named with `-avl` as the source of the variable name-value pairs. Additional (or overriding) variable assignments can be specified with `<var>=<value>` arguments.

The flow may produce some output text and numerical results available as the `NMSEPI_OUTPUT_TEXT` and `NMSEPI_RESULT` environment variables respectively. The output text is produced from the flow using the `@PRINT` directive (see below). If `-trace` is specified, the executed device commands are also added to the output text. This output contains a `#*` prefix for plain commands and `#?` for conditional commands followed by their output and a `#? END`, `#? FAILED`, `#* END`, or `#* FAILED` to indicate the end of of the corresponding command. The output result is produced from the flow by the `@EXIT` directive or the `NMSCmdTerminateCommandFlow` command in callback mode (see below). You can control specific tracing by using the following optional modifiers:

C traces only the executed plain commands. Control construct commands (`@SWITCH`, `@FOREACHPP`, ...) are not traced.

R traces only the plain command responses (also excludes control constructs).

E traces only errored responses from plain commands (also excludes control construct commands).

X traces control construct commands and responses

You can specify one or more such modifier letters following a colon. You can also control tracing from the flow itself by using the `@TRACE` construct.

The output processing command (`NMSCmdGetNumColumns`, `NMSCmdGetColumn`, and `NMSCmdPatternMatch`) can also be used to examine the output.

If `-cb` is specified, the output text is not returned in the `NMSEPI_OUTPUT_TEXT` variable. Instead, the callback bound to the command interface by `NMSCmdBindCallback` is invoked for each line of output (including the `@PRINT` and `#?` and `#*` comments) where it is

available in the usual manner (with reason `RESPONSE`). From the callback, it is possible to force the termination of the flow execution by invoking the `NMSCmdTerminateCommandFlow` command with the desired numerical result (as if `@EXIT` had been reached in the file).

Note that in callback mode, even though the output is returned line by line through the bound callback, the execution of the flow is still synchronous an no other EPI actions will be performed until the flow execution has completed. In other words, the flow/file execution function will not return until the flow execution is complete during which the bound callback will be invoked for each output line. Once the flow execution completes, the bound callback is called with reason `ENDRESP` on success or ERROR on failures, unless the execution was explicitly terminated with `NMSCmdTerminateCommandFlow`.

If `-test` is specified, the command flow/file is executed in test mode as described later.

Command flows consist of a list of device commands with, optionally, substitution variables identified by a dollar sign '`$`' (to specify a plain `$`, escape it as '`\$`'). For example, the following Passport command sets the `committedInformationRate` of a FrameRelay DLCI (read as one line):

```
$name set FrUni/$fruni Dlci/$dlci Sp\
            committedInformationRate $cir
```

A flow containing such a line should be executed with an attribute-value associative array containing at least:

```
set avl(name)=NODER16
```

```
set avl(fruni)=120
set avl(dlci)=25
set avl(cir)=56000
```

Other forms of substitution variable specification include;

${<variable name>**}**	same as without the brackets.
$!**<variable name>, or **${!<variable name>**}**	(strict substitution) When used in device commands (does not apply to the special directives below), the command will be skipped (silently not executed) if the specified variable has no associated value or is empty. Note: This form is only available in actual device commands.
${<variable name>**:-** <default value>**}**	If the specified variable has no associated value or is empty, substitutes the specified default value instead.
$%	This special internal variable holds the contents of the last executed `@SWITCH` command or the value of the matched `\(\)` subpattern of the last executed `@case` command (see below).
$%%	This special internal variable holds the pattern-matched contents (whole or sub-pattern) of the last executed `@IF` command (see below).

$?	This variable contains the numerical result of the last issued macro (@DO) or flow (@INCLUDE/@RUN).
${<variable name>**[**<index>**]}**	This represents an associative array value in the Flow language itself (not to be mistaken by array values in the scripting language). Such values can be directly set (@SET, @DEFINE, @LOCAL) or provided by specialized commands (@FOREACHPP, @SPLITCOMP, @SPLIT). When used, both the variable name and the index can be also be substituted variables (for example, ${array[$i]} in a loop that increments the value of $i). The ! and :- constructs also apply to the array entry (for example, ${array[$i]:-0} defining 0 as the default value for the entry).

Command flows also support special processing directives as indicated in the following table:

@PRINT <string>	appends the specified text (after variable substitution) to the command output (NMSEPI_OUTPUT_TEXT).
@FORMAT <multi-line string...> **@END**	same as @PRINT but for a multi-line piece of text.
@EXIT [<code>]	terminates the flow's execution with the specified result code (NMSEPI_RESULT). If no @EXIT directive is executed, the flow will have its result set to 0 on success or 4 if the flow was terminated by a failed command.

@RETURN [<code>]	like @EXIT but when invoked from an included file (see @INCLUDE), only terminates the execution of the included file and returns to the calling flow/file. If used from within a macro or included/run flow, the return value is available as the $? variable from the calling flow.
@TRY [<command>]	executes the command and ignores its possible failure, even if -bestEffort was not specified). If the command is omitted, sets the current operating mode for subsequent commands as if -bestEffort had been specified. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.
@CRITICAL [<command>] or **@CRIT** [<command>]	executes the command and terminates the flow if the command fails, even if -bestEffort was specified. If the command is omitted, sets the current operating mode to critical for subsequent commands. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.
@TRACE [<command>]	traces this command, even if -trace is not specified. If the command is omitted, sets the current operating mode to trace (as if -trace had been specified) for subsequent commands. Other modifiers (@TRY/@CRITICAL, @LOG/@NOLOG) may also be specified.

@NOTRACE [<command>]	does not trace this command, even if -trace is specified. If the command is omitted, sets the current operating mode to no-trace for the subsequent commands. Other modifiers (@TRY/ @CRITICAL, @LOG/@NOLOG) may also be specified.
@LOG [<command>]	logs this command as long as a log file has been defined (NMSCmdOpenCommandLog or @LOGFILE). If the command is omitted, sets the current operating mode to logging. Other modifiers (@TRY/@CRITICAL, @TRACE/ @NOTRACE) may also be specified.
@NOLOG [<command>]	does not log this command, even if logging was enabled. If the command is omitted, sets the current operating mode to no-logging for the subsequent commands. Other modifiers (@TRY/ @CRITICAL, @TRACE/@NOTRACE) may also be specified.
@LOGFILE [[**+**]<log file path>]]	This controls command logging (see NMSCmdOpenCommandLog). Without arguments, this is equivalent to @LOG. If the log file is identified, the issued commands are logged to it from this point on (unless modified by @NOLOG). If the file path is prefixed with the plus (+) sign, the logs are appended to the file if it exists (else the file is overwritten with the new logs).

```
@IF <var.> [==|!=
            <patterns>]
or
@IF <var.> <|>|<=|>=
            <value
<commands>
...
[@ELSE
<commands>
...]
@END
```

evaluates the specified variable expressions and executes the first command block if it finds that it matches one (==) or does not match any (!=) of the patterns, first form, or compares (numeric or string, as appropriate), the second form, to the specified value. Otherwise, the command block following the @ELSE directive is executed, if any. If just the variable expression is specified, the test is positive if the expanded value is not empty. If the test was a pattern matching one, the value of the sub-pattern is available as the $%% internal variable.

```
@FOR <var> <from> <to>
       [<increment>]
or
@FOR <var> IN
       <token list>
<commands>
...
@END
```

executes the command block repeatedly, in the first form, while incrementing the named (<var>) numerical variable in the AVL from <from> to <to> in jumps of <increment> (defaults to 1), or in the second form, iterating the variable across the list of blank separated tokens. The variable ($<var>) can be used in the command block. The loop can be terminated prematurely by calling the @BREAK command from within the command block. Loops can be nested. The third form allows the variable to scan the existing indicies of the named associative array (for example, the Passport attribute values from @FOREACHPP).

```
@FOREACHPP <var>
        <Passport
         list/display
         command>
<commands>
...
@END
```

executes the specified Passport list command, and then iterates over the returned component names, assigning its name to the named variable and executing the command block. You can terminate the loop by calling the `@BREAK` command from within the command block. Loops can be nested. If the Passport command was a display one, the extracted attribute values are also available as the entries of an associative array of the same name as the specified variable. They are provided in the same way as from the `NMSCmdRecvNextPPComp` function. If wild-cards are used, make sure you specify the -notab display command option.

```
@FOREACHLN <var>
        <command>
<commands>
...
@END
```

executes the specified command command then iterates over the returned output line by line, assigning each one to the named variable and executing the command block. You can terminate the loop by calling the `@BREAK` command from within the command block. Loops can be nested.

@BREAK

invoked from within a `@FOR/ @FOREACHPP/@FOREACHLN/ @WHILE/@WHILDO` loop construct, it terminates the enclosing loop prematurely. In other situations, it acts like `@RETURN` with no return code.

@CONTINUE	invoked from within a @FOR/ @FOREACHPP/@FOREACHLN/ @WHILE/@WHILDO loop construct. This command causes the loop to immediately iterate to the next cycle. Like @BREAK, the following loop-code is not executed but unlike @BREAK, the loop is not abandoned.
@CB <text>	if the Flow is running in callback mode (-cb option), invoke the callback with the specified text as the output text (NMSEPI_OUTPUT_TEXT). The callback reason (NMSEPI_CB_REASON) is then set to FLOW_CALLBACK. The callback may interpret this text as convened and, in reply, set or reset AVL variables with NMSCmdSetFlowCBAVL before returning so the flow can use the results. You can use callback to query another system or a user for a value needed in the flow processing (Wizzard).

```
@SWITCH <test command>
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

executes the test command and executes the commands following the first @CASE whose patterns match the output of the test command. (the optional commands that follow the @SWITCH before the first @CASE are always executed).Only one @CASE block in the @SWITCH construct is executed. @SWITCH blocks can be nested. Patterns are specified in GREP format with a '|' separating alternatives. If no pattern is specified, the @CASE block accepts any output. The @SWITCH command is traced to ouput, if enabled, as:

```
#? <test command>
<command output...>
#? END
```

(If the command could not be executed, the last line will be **#? FAILED** instead).

The @CASE patterns may contain one subexpression (\(\)), each of whose matched value is available in the following code as the $% substitution variable. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @SWITCH.

```
@SWITCHVAL <value expr.>
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

this construct behaves much like @SWITCH but instead of getting its pattern match target value from a device command output, that value is directly specified as a parameter.

```
@WHILE <var.>
       [==|!=|<|>|<=|>=
       <patterns/
value>]
<commands>
...
@END
```

repeatedly executes the command block as long as the text (same as `@IF`) succeeds. The loop can be broken prematurely by invoking `@BREAK`.

```
@INCLUDE <file path>
or
@RUN <file path>
```

the named file is included and processed as though its contents were part of the current flow. The difference betwen `@INCLUDE` and `@RUN` is that variable definitions (`@DEFINE`) performed in the file invoked by `@INCLUDE` apply to the calling flow. Variable definitions in a file invoked by `@RUN` are ignored upon return. If the file is identified as a relative path, the standard Preside Multiservice Data Manager (MDM) search path applies (see above). If the file cannot be read, the flow's execution terminates unless `-bestEffort` was specified or the `@TRY` prefix is used. The usual modifiers (`@TRY`/`@CRITICAL`, `@TRACE`/`@NOTRACE`, `@LOG`/`@NOLOG`) can be specified before `@INCLUDE`/`@RUN`.

@MACRO <name> [<parameters...>] <code text> ... **@ENDMACRO**	defines a new macro that can be executed later with @DO/@IFDO/@WHILEDO. The macro can be given a number of positional argument names to be provided when executed (the last specified parameter name gets all the remaining arguments passed). These parameter names have local scope to the macro (as if defined with @LOCAL). The macro can be recursive. Just like @INCLUDE/@RUN, it can be terminated by @RETURN which specifies a return code available as the $? variable to the caller. Variables defined/set by the macro have the same scope as the caller unless defined with @LOCAL. Macros must be defined before they are used. The macros themselves have global scope and can be redefined.
@DO <macro name> [<arguments...>]	invokes the named defined macro. The specified arguments are assigned (local scope to the macro) to the macro's parameter names -- the name gets the left over arguments). The @RETURNed value from the macro is available as the $? internal variable. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified before @DO.

```
@IFDO <macro name>
      [<arguments...>]
<command>
...
[@ELSE
<commands>
...]
@END
```

merges the functionality of the @IF and @DO constructs. Executes the macro as for @DO then performs either the first command block if the macro returns a 0 result, else performs the second (@ELSE) command block if any. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @IFDO.

```
@WHILEDO <macro name>
      [<arguments...>]
<commands>
...
@END
```

merges the functionality of the @WHILE and @DO constructs. Repeatedly executes the macro as for @DO then the command block as long as the macro returns a 0 result. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @WHILEDO. As with @WHILE, the loop can be broken with @BREAK invoked in the command block.

```
@DEFINE <name> <value>
```

defines or redefines a variable name. The new value is available in the current command block and the blocks it invokes, notably for included files. For example, if a @DEFINE is invoked in a @CASE block, the modified value applies to the commands in that block but not in the commands that follow the @SWITCH construct for that @CASE. Similarly, @DEFINEs used in included files have no effect on the calling command block.

@UNDEFINE <name>	Contrary to @DEFINE, @SET, and @LOCAL, undefines the named variable. All matching associative array values are also undefined. To undefine a single entry in the array, specify its full name (for example, @UNDEFINE <array>[<index>]).
@SET <name> <value1> [**+**|**-**|*****|**/**|**%**|**~** <value2>]	like @DEFINE but sets the variable to the result of the numerical expression (+, -, *, /, % -- remainder). The ~ operator performs a pattern match using the second value as a GREP style pattern pattern list (| separated). The variable is set to the matching portion or to nothing. If the pattern contains a \(\) delineated sub-pattern, it is that matching sub-pattern that is used as the new value. If only the name and first value are specified, the effect is the same as @DEFINE.
@SPLIT[**(**<separators>**)**] <array name> <string>	This tokenizes the specified string. The individual tokens are assigned to indexed elements of the named associative array (starting at 1). The actual variable's value is the number of resulting tokens. By default, tokenization is done on blanks. Alternatively, the separator characters can be provided between brackets. For example, the following will print the individual applications in a Passport AVL, one per line: **@SPLIT(**, \t\n**)** app $avl **@FOR** i 1 $app **@PRINT** ${app[$i]} **@END**

@SPLITCOMP <array name> <component ID>

analyzes the specified component ID and provides the results in the named associative array. The following examples are based on the component EM/TOTO LP/2 Ds1/0).

- **$**<array> the component ID in API format (EM TOTO LP 2 DS1 0)
- **${**<array>**[_MOD]}** the module name (TOTO)
- **${**<array>**[_SUB]}** the subcomponent portion (minus first level) (LP/2 DS1/0)
- **${**<array>**[_PAR]}** the parent subcomp portion (minus last level) (EM/TOTO LP/2)
- **${**<array>**[**<category>**]}** the relative instance value for that level (EM -> TOTO, LP -> 2, DS1 -> 0)

@WAIT <nb seconds>

blocking wait for the specified number of seconds.

```
@ASK <name>
     [:E|:I|:S]
     [/validation
patterns/]
     [=<default value>]
     <prompt string>
@CASK <name>
     [:E|:I|:S]
     [/validation
patterns/]
     [=<default value>]
     <prompt string>
```

asks you (standard input) for the value of the named variable using the specified string as a prompt. The expected type of the value can be specified as one of the following for a plain string:

`:E` for a string token enumeration
`:I` for an integer
`:S` (the default)

You can use a pattern between two forward slashes (//), vertical bars (|). Specify the pattern so that EPI validates the entered value and prompts if there is no match. For enumeration, the pattern is a blank/coma separated list of words to match (for example, `/on off/`). For integers, the pattern is a blank/coma separated list of numeric values or ranges (for example, `/1,3,5,10-15,20/`). For strings (default) the pattern is an extended GREP style pattern list with `|` between alternative patterns (for example, `/.* Ds1\/.*|.* E1\/.*/` -- The forward `/` in the pattern is escaped with a single backwards \ so it is not included as the end of the pattern). If you specify a default value, this value is set to the variable if you enter nothing (carriage return only). If you do not specify a default value, you are prompted when you enter an empty string.

`@CASK` (conditional ask) is similar to `@ASK` except that it does not prompt if the variable already has a non-empty value..

`# <comment>`	comment line.

`<device command>`	actual device command, optionally with embedded variables (`$` prefix) invoked after substitution. If the command fails, the flow execution terminates (no `-bestEffort` nor `@TRY` prefix).
	The command is traced to ouput, if enabled, as:
	`#* <device command>`
	`<command output...>`
	`#* END`
	(If the command indicated an error, the last line will be **`#?`**

Note: Note that `@TRY`, `@CRITICAL`, `@TRACE`, and `@NOTRACE` can be combined. They can also be used with the `@INCLUDE` directive. Also, the command specified with `@SWITCH` can also start with `@TRACE` or `@NOTRACE`.

Example

Assuming a file (examp.tmpl) containing:

```
# Frame Relay QOS example
@SWITCH $name l FrUni/$fruni Dlci/$dlci
@CASE failed|ERROR
  @PRINT $name FrUni/$fruni Dlci/$dlci does not exist!
  @EXIT 1
@CASE
  $name set FrUni/$fruni Dlci/$dlci Sp cir $cir
  $name set FrUni/$fruni Dlci/$dlci Sp bc $bc
  $name set FrUni/$fruni Dlci/$dlci Sp be $be
  @IF $lmi
    $name set FrUni/$fruni Lmi procedures $lmi
    $name set FrUni/$fruni Lmi side $lmiside
  @END
@END
```

This flow can be executed with:

```
if { [ NMSCmdDoCommandFile -avl avl \
                "*" examp.tmpl ] } {
    puts "Failed!!!\n$NMSEPI_OUTPUT_TEXT";
    exit 1;
}
```

If, instead, the contents of the file above are stored/built in a shell variable (for example, `FLOW_STRING`), the flow can then be executed with the following (note the quotes around the `$FLOW_STRING` specification to avoid the shell from absorbing the carriage returns between commands):

```
if { [ NMSCmdDoCommandFlow -avl avl \
                "*" "$FLOW_STRING" ] } {
    puts "Failed!!!\n$NMSEPI_OUTPUT_TEXT";
    exit 1;
}
```

Example

The following are small examples of flow code usage:

```
# determine the Passport version and save it
# for later use
@SWITCH $name d software avl
@CASE base_\([^ ,]*\)
  # $% contains the last \(\) match (the base version)
  @PRINT Passport version : $%
  @DEFINE ppversion $%
  # set variables accordingly (Tm subcomponent
  # introduced?)
  @IF $ppversion == CA.*|CB.*
    @DEFINE tm Tm
  @ELSE
    @DEFINE tm
  @END
@END
...
# use the variable defined above and set the
# p1 parameter to a default value if not set
$name set AtmIf/$atm Vcc/$vpci Vcd $tm txTdp 1 \
                ${p1:-64000}
...
```

```
# create multiple DS1 channels with @FOR
@FOR chan 0 24
  $name add Lp/$lp DS1/$ds1 Chan/$chan
  $name Lp/$lp DS1/$ds1 Chan/$chan timeslots $chan
  # run a secondary flow to create a FrUni
  # for each channel (the flow has access to
  # the current AVL including the $chan variable)
  @RUN addFrUni
@END
```

CAUTION

DoEPITemplate

The DoEPITemplate helps you use and invoke command flows by handling scripting aspects. You only need to create the required flow text. The utility handles everything else, including the Passport configuration pre and post-amble for the configuration flows.

See "DoEPITemplate Utility" (page 607) for the description of the DoEPITemplate utility.

Test Mode

To test command files/flows without executing them, for example, while developing complex configuration templates, you can invoke the command with the `-test` option. This allows you to test the variable substitution, the `@SWITCH`/`@CASE` pattern matching and the general execution flow of the commands with or without actually executing them. In test mode, commands to be excuted are traced to the standard error stream (after variable substitution and prefixed by its line number) then a prompt usually asks for confirmation if it should be executed. The prompt depends on the command being executed:

@FOR <variable name> <from> <to> [<increment>]
or
@FOR <variable name> **IN** <token list>
on each iteration, the prompt offers to execute the command block or not::

```
> Confirm command block execution? ([y]|n|q)>
```

If `q` is answered, the flow's execution is terminated as though an `@EXIT` directive had been encountered. If `y` is answered (the default), the

command block is executed as normal. If n is answered, the loop's execution is terminated as though a @BREAK directive had been encountered.

@SWITCH <test command>
the prompt offers to execute the command or not::

```
> Execute it? (y|[n]|q)>
```

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered, the command is executed as normal. If n is answered (the default), you are then prompted for the output the command would have produced in order to test the @CASE pattern matching:

```
> Command reply? (end with @@)>
```

The output should be terminated with a line containing only the @@ characters. If you do not want to test the pattern matching, enter @@ and you will be prompted for confirmation of the match for each executed @CASE directives.

@CASE [<patterns>]
the prompt indicates if the pattern matching would succeed:

```
> Matches, confirm command block execution?
([y]|n|q)>
```

or fail:

```
> Does not match, execute command block anyways?
(y|[n]|q)>
```

offering to execute the following command block or not.
If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default on success), the command block is executed as normal. If n is answered (the default on failure) the flow is executed as though the @CASE pattern would not match and the next @CASE is block, if any, is tried instead.

@IF <variable> [==|!= <patterns>]
the prompt indicates if the test succeeds:

```
> Test succeeded, confirm command block execution?
([y]|n|q)>
```

or fails:

> Test failed, execute command block anyways? (y|[n]|q)>

offering to execute the following command block or not.
If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default on success), the command block is executed as normal. If n is answered (the default on failure) the flow is executed as though the @IF test failed and the @ELSE is block, if any, is tried instead. If this block is executed, the @ELSE block, if any, will be ignored.

@ELSE
the prompt offers to execute the following block:

> Confirm command block execution? ([y]|n|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command block is executed as normal. If n is answered, the command block is skipped until the corresponding @END construct.

@INCLUDE <command file>, or
@RUN <command file>
the prompt offers to confirm the execution of the file:

> Include/Run the file? ([y]|n|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command file is executed as normal. If n is answered, the command file is not included.

<device command>
the prompt offers to execute the command or not:

> Execute it? (y|[n]|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered, the command is executed as normal. If n is answered (the default) the command is not executed and the next command is tried. The behaviour is the same for commands prefixed with @TRACE, @NOTRACE, @TRY, and @CRITICAL.

```
@PRINT <string>
@EXIT [<code>]
@DEFINE <name> <value>
@END
```

The command is echoed as is without further prompts.

- **NMSCmdSetFlowCBAVL [-iname <name>]**
  <var name> <value> ...

  called within a callback (`NMSCmdBindCallback`). Invoke this command from a command flow executing (`NMSCmdDoCommandFile` or `NMSCmdDoCommandFlow`) by the `@CB` construct. The callback can then inform the flow of the results by setting or resetting some variable assignments. You can specify multiple variable name/value pairs.

- **NMSCmdOpenCommandLog [-iname <name>]**
  [-append]
  <log file path>

  Opens the specified file and logs all commands executed with `NMSCmdSendCommand`, including those issued by a command flow or file. Each device command executed by the flow is logged, not only the flow's execution. If you specify `-append`, the commands are appended to the file if it exists. By default the file is overwritten. The format of the log allows its replay as a command flow or as input to the `cmccmd` utility. The commands are logged from the module on, no group name, and with all variables substituted for flow commands). Note that logging can also be controlled from the flow itself through the `@LOG` and `@LOGFILE` constructs.

- **NMSCmdCloseCommandLog [-iname <name>]**
  Stops command logging and closes the log file opened by `NMSCmdOpenCommandLog`.

Customer Database access

The Customer Database access capabilities or EPI allow you to query and modify the contents of the Customer Databases managed by Preside Multiservice Data Manager (MDM). The Customer Databases are controlled by servers (`cdbserver`) and can be accessed through the Customer Database tool (GUI). Customer Databases can also be accessed by the `cdbextract` and `cdbmerge` utilities, which are more suited for bulk extraction and population. EPI provides more programmability of the database's contents.

For example, by using EPI you can receive an alarm feed from GMDR and extract the matching customer information from the database. You can then use this information to generate a trouble ticket to an external system.

Customer Database Access commands are used in the following sequence:

1 Initialize a Cdb interface.

2 Connect to the Cdb server for the appropriate database.

3 Send Fetch, Store, or Erase commands.

4 Send Query commands and receive the matching replies either synchronously with the RecvReply command or asynchronously by binding a script callback to the Cdb interface.

5 Use Tcl or the provided commands to analyze the Fetch and Query replies.

6 Disconnect from the Cdb server.

7 Drop the Cdb interface.

The Customer Database server is synchronous in that a Cdb interface can only perform one Fetch, Query, Store and Erase command at once. The Fetch, Store, and Erase commands are synchronous in that they always wait for the reply. The Fetch command also provides the matching information on return (similar to a non-wild card Query followed by a RecvReply command).

Customer Database Access commands

The following Customer Database Access commands are provided:

- **NMSCdbInit [-iname <name>]**
  Initializes a Customer Database interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultCdb` to make it the default interface).

- **NMSCdbDrop -iname <name>**
  Drops the named Customer Database interface (frees allocated resources).

- **NMSCdbConnect [-iname <name>] <CDB name> [<CDB server host>]**

  Connects the interface to the CDB server for the indicated database and host (defaults to “localhost”).

  Example

```
NMSCdbInit;
if { [ NMSCdbConnect EastCustDB bcarse88 ] } {
   ... # we’re connected
```

- **NMSCdbDisconnect [-iname <name>]**
  Disconnects the interface from the CDB server. The interface may then be reconnected to another server.

- **NMSCdbFetch [-iname <name>] [-out] [-hier] <component ID>**
  Queries the Customer Database for information on the specified component. The component name is typically specified in canonical format (see `NMSEPIConvertCompId`). The command blocks and waits for the reply which is returned as global variables, and as a result if `-out` is specified. If `-hier` is specified, the routine will look for matching data on the parent components of the specified one if a match on it cannot be found. The `NMSEPI_CDB_COMPID` is also set to the component ID on which the match was found. With `-hier`, both the canonical and display format of the component IDs are searched for in the database. The following environment variables are set:

  `NMSEPI_CDB_COMPID`
  is the component ID.

  `NMSEPI_CDB_RELCOMPID`
  is the associated component ID, if any, an empty string otherwise.

  `NMSEPI_CDB_DATA`
  is the associated textual data.

  `NMSEPI_CDB_DATE`
  is the associated date stamp (as in `YYYYMMDD`).

NMSEPI_CDB_SOURCE
is the associated three character source code.

If `-out` is specified, the matching reply will be output with the following format (matches the output and input to the `cdbextract` and `cdbmerge` utilities):

```
<source:3>;<date:8>;<component ID:65>;<related
component ID:46>;<data length:4>;<data: variable>\n
```

Example

```
if { [ NMSCdbFetch "EM TOTO FRUNI 45" ] } {
   puts "Data: $NMSEPI_CDB_DATA";
   ...
```

If the -out option had been specified, the command would have produced the following string result (the text is split with '\' for readability):

```
EPI;19980508;EM TOTO FRUNI 45                \
                                  ;FRAD AP34    \
                                   ;0024;Client \
Number: AP0002345
```

- **NMSCdbQuery [-iname <name>]**
  -comp <component ID pattern>
  | -rel <component ID pattern>
  | -data <data pattern>

  Queries the Customer Database for information matching the specified grep-style pattern (see `NMSEPIPatternMatch` on page 146). Only one pattern may be specified to match the entry's component name (`-comp`), associated component name (`-rel`), or data (`-data`). The command immediately returns. The returned replies must be extracted synchronously with `NMSCdbRecvReply` or asynchronously through a script callback bound to the interface with `NMSCdbBindCallback`.

  Example

  ```
  if { [ NMSCdbQuery -comp "EM TOTO FRUNI .*" ] }
     ... # query sent successfully
  ```

The following command is supported only in Tcl EPI. It does not function when the graphical Tk extension is used (`NMSCdbRecvReply` must be used instead).

- **NMSCdbRecvReply [-iname <name>] [-out] [<timeout>]**
  Waits for and receives the next Query reply record from the CDB server. A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts as a no-wait poll. If `-out` is specified, the received record is returned as a string result. The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred). The following global variables are set:

  `NMSEPI_CDB_COMPID`
  is the component ID.

  `NMSEPI_CDB_RELCOMPID`
  is the associated component ID, if any, an empty string otherwise.

  `NMSEPI_CDB_DATA`
  is the associated textual data.

  `NMSEPI_CDB_DATE`
  is the associated date stamp (as in `YYYYMMDD`).

  `NMSEPI_CDB_SOURCE`
  is the associated three character source code.

  If `-out` is specified, the matching reply is returned as a string option with the same format as for the `NMSCdbFetch` command.

  `NMSCdbRecvReply` can be called for each reply and will return 0 (also in `NMSEPI_RESULT`) to indicate success. It will return 6 to indicate that no more replies are available. Further calls, as well as calls when there is no active Query command, will return an error indication.

  Example

```
NMSCdbQuery -data “Client Number: AP.*”;
while { [ NMSCdbRecvReply ] } {
   puts “Component: $NMSEPI_CDB_COMPID”;
```

```
puts "Updated:   $NMSEPI_CDB_DATE";
puts "Data:      $NMSEPI_CDB_DATA";
...
```

- **NMSCdbBindCallback [-iname <name>] <callback command>**
  Binds the specified Tcl command string (typically a function invocation with its arguments) to the Cdb interface. This command string will be executed whenever a Query command reply is received from the server for this interface. The following global variables are available to the callback:

  NMSEPI_INAME
  is the API interface name of the callback.

  NMSEPI_CB_REASON
  is the reason for the callback (`LOST_CONNECTION`, `ERROR`, `ENDRESP`, or `RESPONSE`).

  NMSEPI_CDB_COMPID
  is the component ID.

  NMSEPI_CDB_RELCOMPID
  is the associated component ID, if any, an empty string otherwise.

  NMSEPI_CDB_DATA
  is the associated textual data.

  NMSEPI_CDB_DATE
  is the associated date stamp (as in `YYYYMMDD`).

  NMSEPI_CDB_SOURCE
  is the associated three character source code.

  Example

```
proc mycb { } {
   # ...Process the reply...
}
...
NMSCdbQuery -data "AP.*";
```

```
NMSCdbBindCallback mycb;
...
NMSEPIEventLoop; # never returns
```

- **NMSCdbStore [-iname <name>] <component ID> <data>**
  [-rel <related component ID>]
  [-date <date as YYYYMMDD>]
  [-source <source as SSS>]

  Stores the specified information in the Customer Database. Typically, the component name is specified in canonical format (see `NMSEPIConvertCompId` on page 143). The information is added, or replaces that already associated with the component. The component name and data must be specified. An associated component name (`-rel`), a date stamp (`-date`, defaults to the current date), and a 3-character source code (`-source`, defaults to `EPI`) can also be specified. The command is synchronous since it blocks and waits for the reply from the CDB server.

  Example

  ```
  if { [ NMSCdbStore "EM TOTO FRUNI 43" "Client Number:
  DP542
  Contact: (613) 763-2211)" -rel "FRAD 213" -source "GCG"
  ] } {
     ... # data is now stored
  ```

- **NMSCdbErase [-iname <name>] <component ID>**

  Discards the information associated with the specified component in the Customer Database. Typically, the component name is specified in canonical format (see `NMSEPIConvertCompId` on page 143). The command is synchronous since it blocks and waits for the reply from the CDB server.

  Example

  ```
  if { [ NMSCdbErase "EM TOTO FRUNI 52" ] } {
     ... # data is now erased
  ```

Real-Time Alarm Collection

The Real-Time Alarm Collection (RTAC) capabilities of EPI let you query the matching alarms to produce historical alarm reports. RTAC uses the rtaccol server to spool the alarms it retrieves from the GMDR server to workstation files, one file per day (based on the alarm's time stamp). With the

EPI RTAC interface, you can query the spooled alarms between two date-time boundaries. You can also specify filters for any alarm attribute. Specify these filters as GREP-style patterns. The matching alarms are retrieved and their attributes are provided in the same way as with the Alarm&Status specialized API interface.

Use RTAC Access commands in the following sequence:

1 Initialize an RTAC interface.

2 Start a query by specifying its date/time boundaries and attribute-value filters.

3 Fetch the matching alarms.

4 For each fetched alarm, extract the desired attributes and process them.

5 Drop the RTAC interface (usually not done).

The RTAC interface is synchronous because the interface can only perform one query at a time and the Fetch command is blocking.You can limit the amount of time and or the number of alarms the Fetch command scans to support a polling/round-robin form of multi-tasking.

You cannot use the RTAC interface remotely. Use the interface on the same machine that stores (or NFS mounts) the RTAC spool files. The spool files are located using the `RTAC.cfg` configuration file. For details, see Real time alarm collection tool (rtaccol) in 241-6001-310 *Preside MDM Server Reference Guide*.

RTAC Access commands

The following RTAC Access commands are available:

- **NMSRTACInit [-iname=><name>]**
  This command initializes an RTAC interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultRTAC` to make it the default interface).
- **NMSRTACDrop -iname <name>**
  Drops the specified RTAC interface (frees allocated resources).

- **NMSRTACStartQuery [-iname <name>]**
  <start date/time> <end date/time>
  [-filter <attribute name> <pattern>, ...]

  This command initiates an RTAC query for the alarms within the specified start and end date/time. Only alarms whose attributes match the specified filters will be returned. The start time can be specified as a date ("`yyyy mm dd`"), date and time ("`yyyy mm dd hh mm ss`"), as the special values "`0000 00 00`" or "`ANY`" meaning the oldest available alarm, and as the special value "`NOW`" meaning the current (workstation) date and time. Similarly, the end date/time can be specified as a date ("`yyyy mm dd`"), date and time ("`yyyy mm dd hh mm ss`"), as the special values "`9999 99 99`" or "`ANY`" meaning the latest alarm available, or as the special value "`NOW`", meaning the current (workstation) date and time.

  The filter attribute names are the same as those provided by the Alarm and Status API (for details, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*). The filter values are specified as GREP-style patterns for target values similar to the ones output by the Alarm and Status API. Multiple attribute pattern filters can be specified. Filters on the same attribute are ORed together and filters on different attributes are ANDed together (similar to the API rules).

  Examples

```
NMSRTACInit
if { [ NMSRTACStartQuery "ANY" "NOW" \
                -filter event    "set" \
                        severity "critical"
```

```
                        severity “major” ] } {
   ... # ready to fetch all critical/major alarms up
   ... # to now
```

or,

```
NMSRTACInit
NMSEPIConvertTime -api -offset -7200
if { [ NMSRTACStartQuery “$NMSEPI_OUTPUT_TIME”
              “NOW” ] } {
   ... # ready to fetch alarms for the last two hours
```

To initiate a new query, invoke `NMDRTACStartQuery` again.

NMSRTACFetchNextAlarm [-iname <name>] [-out]
[-terse|-normal|-full]
[-var <array name>]
[-timeout <timeout>]
[-maxrecs <max records>]

This command fetches the next alarm matching the criteria specified by `NMSRTACStartQuery` and `NMSRTACAddFilter`. The matching alarms are returned to the `NMSGMDRAPIRecvAlarm` command. The following global variables are set:

NMSEPI_RECORD_TYPE:
is the API record type (always `RESPONSE` in this context).

NMSEPI_ALARM_SEVERITY:
is the severity of the alarm.

NMSEPI_ALARM_EVENT
is the alarm event.

NMSEPI_ALARM_FAULT
is the fault code.

NMSEPI_ALARM_TIME
is the alarm time.

NMSEPI_ALARM_COMPID
is the component ID.

NMSEPI_ALARM_OPDATA
is the operator data.

If you specify the `-out` option, the major attributes are also returned as a string result. If you specify `-terse`, `-normal`, or `-full`, then the `NMSEPI_ALARM_DISPLAY` global variable is set to the corresponding display format (similar to `NMSGMDRAPIFormatAlarm` and `NMSRTACFormatAlarm`) and returned as a string result if `-out` is also specified. If you specify an associative array name with `-var`, the array is filled with all the non-empty attributes of the alarm, the same as for `NMSGMDRAPIRecvAlarm`.
If you use the `-timeout` and/or `-maxrecs` options, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum number of records tested occurs, the command returns the exit code 5 (timeout) as the `NMSEPI_RESULT` variable. This lets you "poll" the RTAC interface for a while and round-robin to other tasks.

Example

```
if { [ NMSRTACStartQuery "ANY" "NOW" \
               -filter event    "set" \
                        severity "critical" \
                        severity "major" ]} {
   while { [ NMSRTACFetchNextAlarm -full
               -var alarm ] } {
        if { $alarm(compCriticality) > 60 } {
               puts "$NMSEPI_ALARM_DISPLAY"
        }
   }
```

If you use the `-timeout` and/or `-maxrecs` options, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum number of records tested occurs, the command returns the exit code 5 (timeout) as the `NMSEPI_RESULT` variable. This lets you "poll" the RTAC interface for a while and round-robin to other tasks.

You can extract the contents of the fetched alarm with the following commands:

`NMSRTACFormatAlarm`, `NMSRTACGetAlarm`, `NMSRTACGetFirstAttribute`, `NMSRTACGetNextAttribute`, and `NMSRTACFindNextAttribute`

- **NMSRTACFormatAlarm [-iname <name>] [-out] terse|normal|full**
  This command produces the last fetched alarm in the specified display format. The output is provided as the `NMSEPI_ALARM_DISPLAY` global variable and as a string result if the `-out` option is specified.

  Example

```
if { [ NMSRTACStartQuery "ANY" "NOW" \
               -filter event    "set" \
                       severity "critical" \
                       severity "major" ] } {
    while { [ NMSRTACFetchNextAlarm -var alarm ] } {
        if { $alarm(compCriticality) > 60 } {
               NMSRTACFormatAlarm full
               puts "$NMSEPI_ALARM_DISPLAY"
        }
    }
}
```

- **NMSRTACGetAlarm [-iname <name>] [-out] [-var <array name>]**
  This command resets the attribute scan to the beginning of the attribute list (for `NMSRTACGetNextAttribute` and `NMSRTACFindNextAttribute`). This command also places the last fetched alarm into the named associative array variable if the `-var` is used and produces a string result if `-out` is used.

- **NMSRTACGetFirstAttribute [-iname <name>] [-out]**
  This command extracts the first attribute of the last fetched alarm. The attribute is provided similarly to the `NMSAPIGetNextAttribute` command.

  `NMSEPI_FIELD_LABEL`
  is the API field type (always"`_attr:"` in this context).

  `NMSEPI_FIELD_NAME`
  is the API attribute name

  `NMSEPI_FIELD_TYPE`
  is the API attribute type element (always "`S`" in this context).

  `NMSEPI_FIELD_VALUE`
  is the API attribute value.

Multiple-line attribute block values are returned as a single multiple-line value.

If you use the `-out` option, the field is returned as a string result in a format that is similar to API. The return error code is in `NMSEPI_RESULT` A value of `6` indicates there are no more fields.

- **NMSRTACGetNextAttribute [-iname <name>] [-out]**
  This command is similar to `NMSRTACGetFirstAlarm` but extracts the next attribute in the list.

  Example
  `NMSRTACGetAlarm`

```
...
while { [ NMSRTACGetNextAttribute ] } {
   if {   "$NMSEPI_FIELD_LABEL" = "_attr:" \
       && "$NMSEPI_FIELD_NAME"  = "time" } {
      puts "Time is: $NMSEPI_FIELD_VALUE"
```

- **NMSRTACFindNextAttribute [-iname <name> [-out]**
  <attribute name>
  This command finds the next named attribute (NRS type name or long name/title) in the attribute list and returns it similarly to `NMSRTACGetFirstAttribute`.

Network Reporting System

The Network Reporting System (NRS) capabilities of EPI let you create device configuration reports on the data stored in the NRS database and its schema (RDF files). You build the NRS query by identifying the node configuration files to scan (by name, pattern, and naming discipline such as keyed and dated) and the component types to extract configuration data from. The configuration data is returned to the EPI script in various ways (for example, environment variable, associative arrays, and standard output). The data values correspond to the way NRS reports currently work. The EPI NRS capabilities also add value by providing program access to the following items:

- NRS schema contents (the RDF files)
- the automatic construction of the component name
- the ability to specify Passport component and attribute types by name and hierarchical path, rather than by numerical IDs

EPI does not provide a means of populating this database. You must use the NRS utilities for this (some examples include nrspop, pnrspop, and sisauto). For more information on the NRS database and its use, see 241-6001-022 *Preside MDM Network Reporting System User Guide*.

Use NRS Access commands in the following sequence:

1 Initialize an NRS interface.

2 Start a query by specifying the target node configurations and the component types to be reported, and/or

3 Load in and interrogate the NRS schema to help create the report (optional).

4 Fetch the matching configuration components. Components are extracted in depth-first order with no garanteed ordering at peer level (as with other NRS reporting mechanisms).

5 For each fetched component, extract the desired attributes and process them.

6 Drop the NRS interface (usually not done).

The NRS interface is synchronous because an interface can only perform one query at a time and the Fetch command is blocking. However, you can limit the amount of time and the number of alarms the Fetch command scans to support a polling/round-robin form of multi-tasking.

You cannot use the NRS interface remotely. Use the NRS interface on the same machine that stores (or NFS mounts) the NRS database. The database and schema are located using the `NRS.cfg` configuration file. For details, see 241-6001-022 *Preside MDM Network Reporting System User Guide*.

- **NMSNRSInit [-iname <name>]**
  Initializes an NRS interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultNRS` to make it the default interface).
- **NMSNRSDrop -iname <name>**
  Drops the specified NRS interface (frees allocated resources).

- **NMSNRSStartQuery [-iname <name>]**
  [[-include] <comp. type>[*|+|^]...]
  [-exclude <comp. type>[*|+|^]...]

  Initiates an NRS query. The query reports the specified component types. Types listed after the `-exclude` option are not reported. By default, no types are reported. Component types are specified as

  `[<device type>/]<component>[*|+|^]`

  where `<device type>` is a recognized NRS device type (`ppc` for Carrier versions of Passports, `ppe` for Enterprise versions, `dpn` for DPN equipment). If no device type is specified, the default type for the installation is taken from the `NRS.cfg` file. The `<component>` specification can be a name for DPN components, a numerical component ID for Passport components, or a name (the full component type name, for example, `FrameRelayUni`, or the prompt, for example, `FrUni`) for Passport components.

  Note: The Passport component names are not unique since there can be multiple Passport components with the same name or prompt. Passport component types are uniquely identified by their numeric component ID. Including or excluding component types to the report by name includes or excludes all the possible matches. This may result in unwanted components being reported.

  Passport components can also be identified as the full hierarchy of full names and/or abbreviations. For example, to include the `ServiceParameter` component of a Frame Relay UNI Dlci, the following component type can be specified: `ppc/EM-FrUni-Dlci-Sp`. In that case, only the specified `Sp` subcomponent (FrUni-Dlci-Sp) will be returned. Sp components of other hierarchies (for example, FrNni-Dlci-Sp) will be ignored.

  The names are not case sensitive. The schema, RDF files, to interpret the names are located as specified in the `NRS.cfg` file.

  If the component type specification uses an asterix (`*`) as a suffix, all the component's possible subcomponents (recursively) are included or excluded, as specified. If you use a plus sign (`+`), only its immediate subcomponents are included or excluded. If you use a caret (`^`) as a suffix, then all the component's possible parent components are included or excluded. If the modifier is applied to a full path Passport component

specification, then only the related components of the specified full path are added.In non-path specifications, all possible parents are included or excluded. You can combine multiple suffixes. It is possible to include whole sub-hierarchies of components then exclude the subcomponents you don't need by using the `-exclude` option. More component types can be include/excluded with the `NMSNRSReportCompType` command. Specify the `-include` option if you want to include more components after the specification of excluded ones.
To start a new NRS query, call `NMSNRSStartQuery` again.

Example(1)

```
NMSNRSInit
if { [ NMSNRSStartQuery ppc/2 ppc/FrameRelayUni* ] } {
    ... # ready to specify the source files for an NRS
    ... # report on the (Carrier) Passport module and
    ... # Frame Relay components (including all its
    ... # subcomponents)
```

Example(2)

```
NMSNRSInit
if { [ NMSNRSStartQuery ppc/EM-AtmIf-Vcc-Vcd^ ] } {
    ... # ready to specify the source files for an NRS
    ... # report on the (Carrier) Atm Virtual Connection
    ... # Description and its indicated parents (Vcc,
    ... # AtmIf and EM).
```

```
NMSNRSReportCompType [-iname <name>]
                                        [[-include]
<comp. type>[*|+|^]...]
                                        [-exclude <comp.
type>[*|+|^]...]
                              |         -list [-out]
```

Includes or excludes additional component types to the report. See `NMSNRSStartQuery` for the specification of the types. `NMSNRSReportCompType` can be called multiple times to include and exclude component types.

This command can also be used to list, with the `-list` option, the component types currently included. These are listed, one per line, as the value of the `NMSEPI_OUTPUT_TEXT` environment variable. If `-out` is specified, they are also returned as a string result.

Note: The reported component types can be manipulated in the middle of a report by invoking the `NMSNRSReportCompType` command as appropriate.

Example

```
if { [ NMSNRSStartQuery ppc/2 ppc/FrameRelayUni* ] } {
   NMSNRSReportCompType -exclude ppc/Signalling \
          ppc/DataLinkConnectionIdentifier*
   ... # Same as previous example except that this time
   ... # the Signalling and DLCI subcomponents (and
   ... # all its subcomponents for the latter) are to
   ... # to be excluded from the report
```

- **NMSNRSAddSource [-iname <name>]**
  -file <file name>
  | -named <device> <module> <name>
  | -keyed <device> <module> <key>
  | -dated <device> <module> <date>
  | -latest <device> <module>
  [<from date/time>]
  | -list [-out]

  This command specifies which NRS data files to report on. The files are located at the path specified in the `NRS.cfg` file. The files are expected to already be there. For details about how to populate the NRS database, see 241-6001-022 *Preside MDM Network Reporting System User Guide*. The file(s) can be specified in a number of ways;

`-file`	Explicitly names the NRS data file to report on (with or without a full path -- the NRS database path as configured in `NRS.cfg` will be used if not specified).
`-named`	Includes NRS data file(s) matching the specified GREP patterns; `<device>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<module>`, the module name, and `<name>`, the configuration file name.

-keyed	Includes NRS data file(s) matching the specified GREP patterns; `<device>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<module>`, the module name, and `<key>`, the configuration file key. Keyed configuration file names have a fixed prefix (the key) followed by a variable two-digit counter. `<key>` only matches the key prefix. For a matching key, the file with the highest two-digit suffix is selected.

-dated	Includes NRS data file(s) matching the specified GREP patterns; <device>, the device type (for example, ppc, ppe, and dpn), <module>, the module name, and <date>, the configuration file date (six digits, not a pattern). Dated configuration file names have a fixed six-digit prefix (the date) followed by a variable two digit counter. <date> only matches the date prefix. For Passports, <date> matches the activation date of the NRS data file name. For both Passport and DPN files, the highest dated file up to the specified date is accepted (and the one with the highest two-digit counter suffix).
-latest	Includes NRS data file(s) matching the specified GREP patterns; <device>, the device type (for example, ppc, ppe, and dpn), and <module>, the module name. If you specify <fromDateTime> as "YYMMDD", "YYYY MM DD", or "YYYY MM DD HH MM SS", only the NRS data files from that date/time forward (UNIX file modifitation time-stamp) are considered. This option is useful for creating incremental reports based on the latest NRS population or the last report invocation. For each matching module, the matching file with the highest file system date is selected (the most recently populated file).

In all cases where multiple files match the patterns provided in the `NMSNRSAddSource` call, only one file per module is selected, that is the highest one in alphanumerical order. For Passport, this also means the one with the highest version counter, the three-digit file suffix. Multiple calls to `NMSNRSAddSource` can select multiple files for the same module.

This command can also be used to list, with the `-list` option, the files currently included in the report. They are listed, one per line, with full path, as the value of the `NMSEPI_OUTPUT_TEXT` environment variable. If you specify the `-out` option, they are also returned as a string result.

Example(1)

This example and the following assume a sample NRS database containing the following files:

```
dpn.R78.4078.R7872.970709.data
dpn.R78.4078.R7888.970715.data
ppc.NODER16.2105.NEWCARD,full,012.001105.data
ppc.NODER16.2105.NEWCARD05,full,001.001102.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
ppc.NODER17.2106.newconf,full,030.010210.data
ppc.NODER17.2106.demo,full,003.010123.data
ppc.EASTOTT.2100.lab,full,005.010123.data
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
ppe.SOUTHRICH.2102.JF,full,003.010124.data
ppe.SOUTHRTP.2113.voice,full,044.010124.data
```

```
if { [ NMSNRSStartQuery dpn/PE^ ] } {
    NMSNRSAddSource -file \
           “dpn.R78.4078.R7872.970709.data“
    ... # ready to fetch module and PE components from
    ... # the specified NRS data file
```

Given the preceding sample database, this call selects the following file:

```
dpn.R78.4078.R7872.970709.data
```

Example(2)

```
if { [ NMSNRSStartQuery ppc/Card ] } {
then
    NMSNRSAddSource -name "ppc" ".*" "NEWCARD.*"
    ... # ready to fetch card components from all
    ... # node configurations whose name start with
    ... # NEWCARD
```

Given the sample database, this call selects following files:

```
ppc.NODER16.2105.NEWCARD,full,012.001105.data
```

(the "latest" of the two matching files for NODER16)

```
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
```

Example(3)

```
if { NMSNRSStartQuery ppc/Card ] } {
    NMSNRSAddSource -keyed "ppc" ".*" "NEWCARD"
    ... # milar as above but for strict KEY syntax
```

Given the sample database, this call selects the following files:

```
ppc.NODER16.2105.NEWCARD05,full,001.001102.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
```

(The other NODER16 file was not selected as its name does not match the syntax of a keyed file)

Example(4)

```
if { [ NMSNRSStartQuery ppc/Card ] } {
    NMSNRSAddSource -dated "ppc" ".*" "010211"
    ... # as above but this time for all node
    ... # configurations dated before or for February
    ... # 11th 2001
```

Given the sample database, this call selects the following files:

```
ppc.NODER16.2105.NEWCARD,full,012.001105.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
ppc.NODER17.2106.newconf,full,030.010210.data
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
```

(All configurations up to 010211 for PPC nodes are taken.)

Example(5)

```
if { [ NMSNRSStartQuery ppc/Software ppe/Software ] } {
    NMSNRSAddSource -latest "ppc" "EAST.*"
    NMSNRSAddSource -latest "ppe" "SOUTH.*"
    ... # ready to fetch Software components from the
```

```
... # latest configuration of the nodes whose names
... # start with EAST or SOUTH
```

Given the sample database, this call selects the following files:

```
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
ppe.SOUTHRICH.2102.JF,full,003.010124.data
ppe.SOUTHRTP.2113.voice,full,044.010124.data
```

(Assuming the UNIX file data matches the ordering of the activation dates in the file names.)

- **NMSNRSFetchNextComponent [-iname <name>] [-out] [-var <array name> [-byname]] [-skip <level>] [-stop <level>] [-timeout <timeout>] [-maxrecs <maxrecs>] [-marker <seek marker>]**

  Fetches the next matching component (according to the component types specified in `NMSNRSStartQuery` and `NMSNRSReportCompType`) from the selected (through `NMSNRSAddSource`) NRS data files. The matching component information is available through the following global variables:

  `NMSEPI_NRS_COMPID`
  is the full Component ID of the fetch component (the component is specified in mixed case, with space separators, for example, "`EM NODER16 FrUni 132 Dlci 206 Sp $`", and can be manipulated with `NMSEPIConvertCompId`).

  `NMSEPI_NRS_COMPTYPE`
  is the component type of the fetched component (the component type is specified as `<device>/<type>` where `<device>` is the NRS device type (`ppc`, `ppe`, or `dpn`), and `<type>` is the component type (a name for DPN, a numerical component ID for Passport, for example, `dpn/PE` or `ppe/8664`).

  `NMSEPI_NRS_COMPTITLE`
  is the (long) type name of the fetched component, for example, `ServiceParameterProv` for the component identified above.

  `NMSEPI_NRS_COMPABBREV`

is the short type name of the fetched component (for Passport, it is the prompt), for example, `Sp` for the component identified above.

`NMSEPI_NRS_VALUE`
is the instance value of the fetched component (the value of its most specific component ID category/value pair, for example, "`$`" for the component identified above).

`NMSEPI_NRS_LEVEL`
is the hierarhcy level of the fetched component (starting at 0 for the module level (for example, for the component identified above the level is 3). Note that matching components are returned in depth-first order.

`NMSEPI_NRS_FILE_PATH`
is the NRS data file from which the matching component was fetched.

If you specify the `-out` option, the first four items above are also output, one per line, as a string result. If you use the `-var` option to name an associative array, the array is filled with the NRS component description. The array entry index is the attribute type, as specified in NRS, names for DPN and the special values "`OAM`", "`_COMPONENT"`, "`_HIERARCHY_LEVEL"`, and a numerical attribute ID for Passport components (for example, `16567` for the `Frame Relay Sp committedInformationRate` attribute). If you specify the `-byname` option, the index is indicated except for the special values indicated, which are then reported as "`Ownership`", "`Component"` and "`Hierarchy_level`" respectively, and for Passport components for which the full attribute name is used instead (for example, "`committedInformationRate`", not "`cir`"). The entry value is the corresponding attribute value. EPI also includes special entries with `_EPI_COMPID`, `_EPI_TITLE`, and `_EPI_ABBREV` index containing the information corresponding to the `NMSEPI_NRS_COMPID`, `EPI_NRS_COMPTITLE`, and `EPI_COMPABBREV` variables above.

The individual attributes of the fetched component can also be extracted one at a time using the `NMSNRSGetFirstAttribute`, `NMSNRSGetNextAttribute`, and `NMSNRSFindAttribute` commands.

If you use the `-marker` option, the specified parameter must be a component marker previously saved from a call to `NMSNRSGetComponentMarker`. This repositions the NRS file scanning to the specified marker before fetching the next matching component. If the same component type specifications are used, the same component whose marker is saved is returned. See the description of `NMSNRSGetComponentMarker` for more information. If the specified marker is an empty string, the option is ignored (as if it was not specified).

A marker may refer to another file than the one being scanned. In this case, the marked file is pushed onto the interface and the current file is re-scanned from the top when the marked file is fully processed.

If you use the `-skip` option to specify a hierarchy level, the command returns the next matching component of a hierachy level that is smaller or equal to the one given. Other matching components of a higher level are ignored. This restricts the search by skipping over components that undesirable (for example, if a Service Parameter (level 3) component attribute of a Passport Frame Relay DLCI component (level 2) doe not meet the necessary criteria, the next call to `NMSCmdFetchNextComponent` can specify `-skip 2` which skips to the next DLCI component, ignoring any intervening matches. NRS components are delivered in depth-first order.

Note: Specifying `-skip 0` is an efficient way of skipping an entire module (configuration file) since EPI does not try to match the remaining components. The command tries to match components in the next module-configuration specified with `NMSNRSAddSource`. Specifying `-skip -1` has the same effect as not specifying the option at all.

If you use the `-stop` option, the next matching component is returned as normal. In addition, if a non-matching component of the specified level or a lower value is found, the command returns with an empty component (all variables are empty except for `NMSEPI_NRS_LEVEL`). As a result, you receive information when the member of a component sub-tree is scanned and another sub-tree of the same level is about to be scanned (the new sub-tree matches, then it is returned as normal, else the empty component is returned as a form of component terminator). For

example, assuming `FrUnis` (level 1) are being extracted with `-stop 1` specified as soon as the first one is found, all matching `FrUnis` will be returned as normal and an empty component is returned once a `Vs` components (also at level 1 but not requested by the query) is found. This technique allows one to maintain a state machine that knows when not to expect more sub-components of a specific sub-tree.

If you set the `-timeout` or `-maxrecs` options, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum number of records tested occurs, the command returns the exit code `5` (timeout) as the `NMSEPI_RESULT` variable. This lets you "poll" the NRS interface for a while and round-robin to other tasks.

Example(1)

```
# looking for FrUnis with no running LMIs
...
NMSNRSStartQuery ppc/FrUni+
...
set skip -1
while { [ NMSNRSFetchNextComponent -skip $skip \
          -var comp -byname ] } {
   set skip -1
   if {   $NMSEPI_NRS_COMPABBREV == "Lmi" \
       && $comp(procedures) == "none" } {
          puts "$NMSEPI_NRS_COMPID has no active Lmi"
   } else {
          # skip to the next FrUni
          set skip [ expr $NMSEPI_NRS_LEVEL - 1 ]
   }
}
```

Example(2)

```
...
NMSNRSStartQuery ppc/ServiceParametersProv
...
NMSNRSFetchNextComponent -var comp -byname
```

can produce the following values for the `comp` array:

```
  comp(_EPI_COMPID)=\
          "EM NODER16 FrUni 132 Dlci 206 Sp $"
```

```
comp(_EPI_TITLE)=ServiceParametersProv
comp(_EPI_ABBREV)=Sp
comp(Component)=ppe/8664
comp(Hierarchy_level)=3
comp(Ownership)=OWNER_IWS
comp(EM)=NODER16
comp(FrameRelayUni)=132
comp(DataLinkConnectionIdentifier)=206
comp(ServiceParametersProv)=$
comp(committedInformationRate)=64000
comp(measurementInterval)=0
comp(committedBurstSize)=64000
comp(rateEnforcement)=on
comp(rateAdaptation)=off
comp(excessBurstSize)=0
comp(maximumFrameSize)=2100
comp(accounting)=on
comp(updateBCI)=off
comp(raSensitivity)=7
```

without the `-byname` option, the output is:

```
comp(_EPI_COMPID)=\
        "EM NODER16 FrUni 132 Dlci 206 Sp $"
comp(_EPI_TITLE)=ServiceParametersProv
comp(_EPI_ABBREV)=Sp
comp(_COMPONENT)=ppe/8664
comp(_HIERARCHY_LEVEL)=3
comp(OAM)=OWNER_IWS
comp(_2)=NODER16
comp(_279)=132
comp(_302)=206
comp(_8664)=$
comp(8668)=on
comp(8670)=64000
comp(8669)=64000
comp(5997)=off
comp(8671)=0
comp(8672)=0
comp(8673)=off
comp(8674)=7
comp(8675)=on
comp(8667)=2100
```

- **NMSNRSGetComponentMarker [-iname <name>] [-out]**
  Extracts a marker for the current fetched component and returns it as the `NMSEPI_NRS_COMP_MARKER` environment variable or on standard output if `-out` is used. The marker may be saved and used later as an argument to `NMSNRSFetchNextComponent's -marker` option to rescan a component structure (not the whole file) which works around the NRS peer component type that is not specified (all instances of a subcomponent type X are together, but it is not specified if instances of type X appear before or after those of its peer type Y). For example, an NRS interface may be used to scan for Frame Relay interfaces and their DNA sub-components. When an FrUni is found, its marker is saved and the interface is used to locate its Dna. Then the same (requires one to manipulate the selected types with `NMSNRSReportCompType`) or another NRS interface can be used to scan for the FrUni's Dlci subcomponents. The marker may be exchanged between properly initialized NRS interfaces. markers may also refer to different files. If the marker cannot be computed, an empty string is returned.

  Example

```
# Assume two NRS interfaces (inames fruni and dlci)
# configured to scan for FrUnis/Dnas and Dlcis
# respectively
while { [NMSNRSFetchNextComponent -iname fruni \
          -var comp] } {
   if { $NMSEPI_NRS_COMP_ABBREV == "FrUni" ) {
          # ... process FrUni information...
          NMSNRSGetComponentMarker
          set marker "$NMSEPI_NRS_COMP_MARKER"
   } else if { $NMSEPI_NRS_COMP_ABBREV == "Dna" } {
          # ... process DNA information ...
          set stopAt -1
          while { [ NMSNRSFetchNextComponent \
                          -iname dlci -stop $stopAt \
                          -marker "$marker"] } {
             set marker ""
             set stopAt 1
             # ... process the DLCIs ...
          }
   }
}
```

- **NMSNRSGetComponent [-iname <name>] [-out] [-var <array name> [-byname]]**

  Resets the fetched component's attribute list for the `NMSNRSGetFirstAttribute`, `NMSNRSGetNextAttribute`, and `NMSNRSFindAttribute` commands and returns the description of the component similarly to `NMSNRSFetchNextComponent`.

- **NMSNRSGetFirstAttribute [-iname <name>] [-out] [-var <array name>]**

  Extracts the first attribute from the fetched component. The following global variables are set with the description of the attribute:

  `NMSEPI_FIELD_LABEL`
  is the attribute's name (for Passport, it is the attribute's full name, not its prompt, for example, "`Percent heap`" for DPN, "`committedInformationRate`" for Passport).

  `NMSEPI_FIELD_NAME`
  is the attribute's type; a string for DPN, a numeric attribute ID for Passport (for example, "`LOADPETYPE`" for DPN, `8669` for Passport).

  `NMSEPI_FIELD_TYPE`
  is the NRS type of the attribute; `BIT_STRING`, `BOOLEAN`, `DNA`, `HEXADECIMAL`, `INVISIBLE`, `LISTINDEX`, `NOKEY`, `NUMERIC`, or `STRING`.

  `NMSEPI_FIELD_VALUE`
  is the attribute's value.

  If `-out` is specified, the values are also returned, one line in the same order as above, as a string result.
  If `-var` is used to name an associative array, that array is filled with the attribute's description with entries with the following indicies:

  `name`
  the attribute's type (same as `NMSEPI_FIELD_NAME` above).

  `value`
  the attribute's value (same as `NMSEPI_FIELD_VALUE` above).

  `type`

the attribute's NRS type (same as `NMSEPI_FIELD_TYPE` above).

`title`
the attribute's full name (same as `NMEPI_FIELD_LABEL` above).

`abbrev`
for DPN, it is the same as the `title` entry, for Passport it is the attribute's prompt.

`width`
the attribute's maximum width in NRS.

- **NMSNRSGetFirstAttribute [-iname <name>] [-out] [-var <array name>]**
  Extracts the next attribute from the fetched component. The attribute information is returned similarly to `NMSNRSGetFirstAttribute`.

- **NMSNRSFindAttribute [-iname <name>] [-out] [-var <array name>] <name>**
  Extracts the named attribute from the fetched component. The search is case-insensitive. For Passport, both the full attribute name and the prompt can be used. The attribute information is returned in a similar manner to `NMSNRSGetFirstAttribute`.

  Example

```
# looking for FrUnis with no running LMIs (as above)
...
NMSNRSStartQuery ppc/EM-FrUni-Lmi
...
set skip -1
while { [ NMSNRSFetchNextComponent -skip $skip ] } {
    set skip -1
    NMSNRSFindAttribute "procedures"
    if { $NMSEPI_FIELD_VALUE == "none" } {
           puts "$NMSEPI_NRS_COMPID has no active Lmi"
    } else {
           # skip to the next FrUni
           set skip [ expr $NMSEPI_NRS_LEVEL - 1 ]
    }
}
```

- **NMSNRSGetComponentSchema [-iname <name>] [-out] [-var <array name> [-byname]] [<comp type>[*|+|^]]**

  Use this command to examine the component schema for the current fetched component (from `NMSNRSFetchNextComponent`) when no `<comp type>` is specified. You can also use it to examine the NRS schema (RDFs) in general when `<comp type>` is specified in a similar manner to the `NMSNRSStartQuery` command, Use the `*`, `+`, or `^` prefix to automatically load the RDFs for the subcomponents and parents of the specified component respectively. (Note that components loaded with this command are not necessarily reported. Use `NMSNRSStartQuery` or `NMSNRSReportCompType` for this).

  Note: For Passport, `<comp type>` can also be specified as the component's numerical ID, name, prompt or full name/prompt path. Passport component names and prompts are not unique since there can be multiple Passport components with the same name or prompt (but different numerical IDs). It is not determined which matching component will be returned by `NMSNRSGetComponentSchema` in that case.

  The following global variables are set by this command:

  `NMSEPI_NRS_SCHEMA_NAME`
  is the component's type specified as `<device>/<comp>` where `<device>` is the device type (`ppc`, `ppe`, or `dpn`), and `<comp>` is the component type (a name for DPN, a numerical ID for Passport, for example, `dpn/UTP` for DPN, `ppc/302` for Passport).

  `NMSEPI_NRS_SCHEMA_TITLE`
  is the component's name (a name for DPN, the full component name, for Passport, for example, "`UTP`" for DPN, "`FrameRelayUni`" for Passport).

  `NMSEPI_NRS_SCHEMA_ABBREV`
  is the component's abbreviation (a name for DPN, the prompt for Passport, for example, "`UTP`" for DPN, "`FrUni`" for Passport).

  `NMSEPI_NRS_SCHEMA_OWNERS`
  is the list of the component's direct parent component types. The types are on a single line, space separated, and specified as `<device>/`

<comp> similarly to NMSEPI_NRS_SCHEMA_NAME above. For example, the owners for dpn/UTP are "dpn/PO", and for ppc/VritualFramer they are "ppc/9200 ppc/1543 ppc/279 ppc/3742".

NMSEPI_NRS_SCHEMA_SUBORDINATES
is the list of the component's direct sub-component types. The types are on a single line, space separated, and specified as <device>/<comp> similarly to NMSEPI_NRS_SCHEMA_NAME above. For example, for dpn/UTP the subordinates are "dpn/UTPLINK dpn/UTPLINK_EXT dpn/MI8_CUS_MODEM_PROF_ENV dpn/UTPDIALUP dpn/UTPPASSWD_ENV" , and for ppc/DataLinkConnectionIdentifier they are "ppc/8664 ppc/219 ppc/16565".

NMSEPI_NRS_SCHEMA_ATTRS
is the list of the component's attribute types. The types are on a single line, space separated, and specified as names for DPN and as numerical IDs for Passport. For example, for dpn/UTP, the attributes are "_COMPONENT _HIERARCHY_LEVEL _PM _PE _PI _PO _UTP OAM", and for ppc/FrameRelayUni they are "_COMPONENT _HIERARCHY_LEVEL _2 _279 OAM 96 3177 3175 9332".

If you specify the -out option, the component type, (full) name, abbreviation (prompt), list of direct parent component types, list of direct subcomponent types, and list of attributes are returned as a string result one per line (the last two lists are on a line each, with space separators between the types).

If you use the -var option to name an associative array, the command fills two arrays with the schema information. The <array name> array contains the following entries indicated by their index:

name
is the component type (same as NMSEPI_NRS_SCHEMA_NAME above).

title
is the component name (same as NMSEPI_NRS_SCHEMA_TITLE above).

abbrev
is the component abbreviation (same as NMSEPI_NRS_SCHEMA_ABBREV

above).

owners
is the component list of direct parent types (same as NMSEPI_NRS_SCHEMA_OWNERS above).

subordinates
is the component list of direct subcomponent types (same as NMSEPI_NRS_SCHEMA_SUBORDINATES above).

attrs
is the component attribute type (same as NMSEPI_NRS_SCHEMA_ATTRS above).

The <array name>_fields array contains the following entries for each attribute types supported by the component (where <field type> is the attribute name for DPN and the attribute numeric ID for Passport or the attributes full name/title if -byname is specified).

<field type>,**name**
is the indexed field NRS attribute type name.

<field type>,**title**
is the indexed field name (a string for DPN, the full attribute name for Passport).

<field type>,**type**
is the indexed field NRS type (see the description of NMSEPI_FIELD_TYPE for NMSNRSGetFirstAttribute).

<field type>,**width**
is the indexed field maximum NRS width.

<field type>,**abbrev**
is the indexed field abbreviation (same as the title for DPN, the attribute prompt for Passport).

<field type>,**group**
is the indexed field attribute group type .

Example(1)

```
# Lists (by full name) the attributes of the Passport
# Lp component
NMSNRSGetComponentSchema -var comps \
          ppc/LogicalProcessor
puts "$NMSEPI_NRS_SCHEMA_TITLE"
# print its attributes by name
foreach i $NMSEPI_NRS_SCHEMA_ATTRS {
   puts " $comps_fields($i,title)"
}
```

Example(2)

```
NMSNRSInit
NMSNRSGetComponentSchema -var comp dpn/UTP
```

can produce the following values for the comp array:

```
comp(name)=dpn/UTP
comp(title)=UTP
comp(abbrev)=UTP
comp(owners)=dpn/PO
comp(subordinates)="dpn/UTPLINK dpn/UTPLINK_EXT \
        dpn/MI8_CUS_MODEM_PROF_ENV dpn/UTPDIALUP \
        dpn/UTPPASSWD_ENV"
comp(attrs)="_COMPONENT _HIERARCHY_LEVEL _PM _PE \
        _PI _PO _UTP OAM"
```

and the following values for the comp_fields array:

```
comp_fields(_COMPONENT,abbrev)=Component
comp_fields(_COMPONENT,group)=
comp_fields(_COMPONENT,name)=_COMPONENT
comp_fields(_COMPONENT,title)=Component
comp_fields(_COMPONENT,type)=STRING
comp_fields(_COMPONENT,width)=0
comp_fields(OAM,abbrev)=Ownership
comp_fields(OAM,group)=
comp_fields(OAM,name)=OAM
comp_fields(OAM,title)=Ownership
comp_fields(OAM,type)=STRING
comp_fields(OAM,width)=10
comp_fields(_HIERARCHY_LEVEL,abbrev)=\
        Hierarchy_level
comp_fields(_HIERARCHY_LEVEL,group)=
```

```
comp_fields(_HIERARCHY_LEVEL,name)=\
        _HIERARCHY_LEVEL
comp_fields(_HIERARCHY_LEVEL,title)=\
        Hierarchy_level
comp_fields(_HIERARCHY_LEVEL,type)=NUMERIC
comp_fields(_HIERARCHY_LEVEL,width)=2
comp_fields(_PM,abbrev)=PM
comp_fields(_PM,group)=
comp_fields(_PM,name)=_PM
comp_fields(_PM,title)=PM
comp_fields(_PM,type)=STRING
comp_fields(_PM,width)=12
comp_fields(_PE,abbrev)=PE
comp_fields(_PE,group)=
comp_fields(_PE,name)=_PE
comp_fields(_PE,title)=PE
comp_fields(_PE,type)=NUMERIC
comp_fields(_PE,width)=2
comp_fields(_PI,abbrev)=PI
comp_fields(_PI,group)=
comp_fields(_PI,name)=_PI
comp_fields(_PI,title)=PI
comp_fields(_PI,type)=NUMERIC
comp_fields(_PI,width)=2
comp_fields(_PO,abbrev)=PO
comp_fields(_PO,group)=
comp_fields(_PO,name)=_PO
comp_fields(_PO,width)=2
comp_fields(_PO,type)=NUMERIC
comp_fields(_PO,title)=PO
comp_fields(_UTP,abbrev)=UTP
comp_fields(_UTP,group)=
comp_fields(_UTP,abbrev)=_UTP
comp_fields(_UTP,title)=UTP
comp_fields(_UTP,type)=NOKEY
comp_fields(_UTP,width)=1
```

Example(3)

```
NMSNRSInit
NMSNRSGetComponentSchema -var comp ppc/FrameRelayUni
```

can produce the following values for the `comp` array:

```
comp(name)=ppc/279
```

```
comp(title)=FrameRelayUni
comp(abbrev)=FrUni
comp(owners)=ppc/2
comp(subordinates)=”ppc/161 ppc/16502 ppc/287 \
        ppc/10990 ppc/586 ppc/9314 ppc/302 \
        ppc/10649 ppc/8600 ppc/1768”
comp(attrs)=”_COMPONENT _HIERARCHY_LEVEL _2 _279 \
        OAM 96 3177 3175 9332”
```

and the following values for the comp_fields array:

```
comp_fields(_COMPONENT,abbrev)=Component
comp_fields(_COMPONENT,group)=Component
comp_fields(_COMPONENT,name)=_COMPONENT
comp_fields(_COMPONENT,title)=Component
comp_fields(_COMPONENT,type)=STRING
comp_fields(_COMPONENT,width)=0
comp_fields(OAM,abbrev)=Ownership
comp_fields(OAM,group)=
comp_fields(OAM,name)=OAM
comp_fields(OAM,title)=Ownership
comp_fields(OAM,type)=STRING
comp_fields(OAM,width)=10
comp_fields(_HIERARCHY_LEVEL,abbrev)=\
        Hierarchy_level
comp_fields(_HIERARCHY_LEVEL,group)=
comp_fields(_HIERARCHY_LEVEL,name)=\
        _HIERARCHY_LEVEL
comp_fields(_HIERARCHY_LEVEL,title)=\
        Hierarchy_level
comp_fields(_HIERARCHY_LEVEL,type)=NUMERIC
comp_fields(_HIERARCHY_LEVEL,width)=2
comp_fields(_2,abbrev)=EM
comp_fields(_2,group)=
comp_fields(_2,name)=_2
comp_fields(_2,title)=EM
comp_fields(_2,type)=STRING
comp_fields(_2,width)=12
comp_fields(_279,abbrev)=FrUni
comp_fields(_279,group)=
comp_fields(_279,name)=_279
comp_fields(_279,title)=FrameRelayUni
comp_fields(_279,type)=NUMERIC
comp_fields(_279,width)=5
```

```
comp_fields(3175,abbrev)=ifI
comp_fields(3175,group)=IfEntryProv
comp_fields(3175,name)=3175
comp_fields(3175,title)=ifIndex
comp_fields(3175,type)=NUMERIC
comp_fields(3175,width)=5
comp_fields(3177,abbrev)=ifState
comp_fields(3177,group)=IfEntryProv
comp_fields(3177,name)=3177
comp_fields(3177,title)=ifAdminStatus
comp_fields(3177,type)=STRING
comp_fields(3177,width)=7
comp_fields(96,abbrev)=cid
comp_fields(96,group)=CustomerIdentifierData
comp_fields(96,name)=96
comp_fields(96,title)=customerIdentifier
comp_fields(96,type)=STRING
comp_fields(96,width)=4
comp_fields(9332,abbrev)=numberOfEmissionQs
comp_fields(9332,group)=EmissionPriorityQs
comp_fields(9332,name)=9332
comp_fields(9332,title)=numberOfEmissionQs
comp_fields(9332,type)=STRING
comp_fields(9332,width)=1
```

If -byname is specified, the fields report as follows:

```
comp_fields(Component,abbrev)=Component
comp_fields(Component,group)=Component
comp_fields(Component,name)=_COMPONENT
comp_fields(Component,title)=Component
comp_fields(Component,type)=STRING
comp_fields(Component,width)=0
comp_fields(Ownership,abbrev)=Ownership
comp_fields(Ownership,group)=
comp_fields(Ownership,name)=OAM
comp_fields(Ownership,title)=Ownership
comp_fields(Ownership,type)=STRING
comp_fields(Ownership,width)=10
comp_fields(Hierarchy_level,abbrev)=\
        Hierarchy_level
comp_fields(Hierarchy_level,group)=
comp_fields(Hierarchy_level,name)=\
        _HIERARCHY_LEVEL
```

```
comp_fields(Hierarchy_level,title)=Hierarchy_level
comp_fields(Hierarchy_level,type)=NUMERIC
comp_fields(Hierarchy_level,width)=2
comp_fields(EM,abbrev)=EM
comp_fields(EM,group)=
comp_fields(EM,name)=_2
comp_fields(EM,title)=EM
comp_fields(EM,type)=STRING
comp_fields(EM,width)=12
comp_fields(FrameRelayUni,abbrev)=FrUni
comp_fields(FrameRelayUni,group)=
comp_fields(FrameRelayUni,name)=_279
comp_fields(FrameRelayUni,title)=FrameRelayUni
comp_fields(FrameRelayUni,type)=NUMERIC
comp_fields(FrameRelayUni,width)=5
comp_fields(ifIndex,abbrev)=ifI
comp_fields(ifIndex,group)=IfEntryProv
comp_fields(ifIndex,name)=3175
comp_fields(ifIndex,title)=ifIndex
comp_fields(ifIndex,type)=NUMERIC
comp_fields(ifIndex,width)=5
comp_fields(ifAdminStatus,abbrev)=ifState
comp_fields(ifAdminStatus,group)=IfEntryProv
comp_fields(ifAdminStatus,name)=3177
comp_fields(ifAdminStatus,title)=ifAdminStatus
comp_fields(ifAdminStatus,type)=STRING
comp_fields(ifAdminStatus,width)=7
comp_fields(customerIdentifier,abbrev)=cid
comp_fields(customerIdentifier,group)=\
        CustomerIdentifierData
comp_fields(customerIdentifier,name)=96
comp_fields(customerIdentifier,title)=\
        customerIdentifier
comp_fields(customerIdentifier,type)=STRING
comp_fields(customerIdentifier,width)=4
comp_fields(numberOfEmissionQs,abbrev)=\
        numberOfEmissionQs
comp_fields(numberOfEmissionQs,group)=\
        EmissionPriorityQs
comp_fields(numberOfEmissionQs,name)=9332
comp_fields(numberOfEmissionQs,title)=\
        numberOfEmissionQs
comp_fields(numberOfEmissionQs,type)=STRING
comp_fields(numberOfEmissionQs,width)=1
```

Sample DtKsh scripts

This section provides three sample Tcl EPI scripts: Expect paging (API interface), Passport Card inventory (Command interface), and Passport DLCI configuration report (NRS interface).

Expect paging script sample

The following example combines the Expect extension of Tcl with Tcl EPI to produce a script that pages an operator when a specified alarm is received by Preside Multiservice Data Manager (MDM). Expect is required since the program used to communicate with the pager through a modem (`tip`) is TTY based. This example is available in the Preside Multiservice Data Manager (MDM) load in `/opt/MagellanNMS/cfg/macros/nms/src/PageIt.expect`.

This is a simple example of network operations automation in which certain major events are automatically dispatched to the proper operators (notably during off-hours).

```
#!/bin/sh
# The next line restarts using Expect.
exec expect "$0" "$@"

# Load the Tcl EPI extensions.

source /opt/MagellanNMS/lib/nmsepi.tcl;


# Global settings (to be customized).

set TIPDEVICE hardwire;

set ATCOMMAND atdt12345678;


# This procedure uses Tip to call a pager.

proc doTip {} {
    global TIPDEVICE;

    global ATCOMMAND;


    exp_spawn "/bin/tip" $TIPDEVICE;

    expect {
```

```
            "connected" {
                exp_send "$ATCOMMAND\n";
                expect {
                    -re "$ATCOMMAND\r\n" {
                        exp_continue;
                    }
                    -re "^\r\n" {
                        exp_send "~.\r";
                    }
                    eof {
                        return;
                    }
                }
            }
            eof {
                return;
            }
        }
        exp_close;
}

# Initializes the Alarm and Status API interface and
# creates the alarm sieve.
NMSGMDRAPIInit;
NMSGMDRAPIConnect -ssel;
NMSAPIRegister pageit;
if { [ NMSGMDRAPICreateAlarmSieve -attr \
"_attr: eventFilter SS faultCode LEFT S 4001" \
"_attr: eventFilter SS event EQ E SET" \
```

```
"_attr: eventFilter SS severity EQ E major" \
"_attr: eventFilter SS severity EQ E critical" ] != \
1 } {
    exit 1;
}

# For each received alarm:
while { [ NMSGMDRAPIRecvAlarm ] } {
    set f [open "/opt/MagellanNMS/data/PagedAlarms" "a"
]
    puts $f
"$NMSEPI_ALARM_TIME\t$NMSEPI_ALARM_EVENT\t\
$NMSEPI_ALARM_SEVERITY\t$NMSEPI_ALARM_FAULT\
\t$NMSEPI_ALARM_COMPID";
    close $f;
    doTip;
}
```

Passport Card inventory sample

The following Tcl script uses the command interface to produce a card inventory of a specified Passport node (using the current user session). This example is available in the Preside Multiservice Data Manager (MDM) load in `/opt/MagellanNMS/cfg/macros/nms/src/PPCardInv.tcl`.

```
#!/bin/sh
# The next line restarts using tclsh. \
exec tclsh "$0" "$@"
# Load the Tcl EPI extensions.
source /opt/MagellanNMS/lib/nmsepi.tcl;

# Extract the arguments.
if { $argc < 2 } {
    puts "ppcardrep <group> <passport>";
```

```
    exit 1;
}
set grp [lindex $argv 0]
set mod [lindex $argv 1]

# Initializes the command interface.
NMSCmdInit;
if { [ NMSCmdConnect ] != 1 } {
    puts “Failed to connect to the command Session \
    servers.”;
    exit 1;
}

puts “
Passport Card Inventory
-----------------------------------------------------\
------------------
Node        Card Type        Inserted    Serial #      \
Firm. Rev.     LP
---------- ---- ---------- ----------- ------------- \
-------------- ------”;

# Do we have access to the node.
NMSCmdSendCommand $grp “$mod h -v(d) shelf card”;
NMSCmdRecvFullReply;
if { [NMSCmdPatternMatch “Shelf Card”] != 1 } {
    puts “
*** Passport node $mod does not seem to be reachable.
“;
```

```
    exit 1;
}
if { [NMSCmdPatternMatch "noTabular"] } {
    set opt "-noTabular";
} else {
    set opt ""
}

# Get the number of slots from the shelf component.
NMSCmdSendCommand $grp "$mod d $opt shelf \
numberOfSlots";
while { [NMSCmdRecvNextLine] } {
    if { [NMSCmdGetColumn 1 "numberOfSlots"] } {
        set numberOfSlots [NMSCmdGetColumn -out 3];
    }
}

# Get the needed attributes from all card components.
set cardType ""
NMSCmdSendCommand $grp "$mod d $opt shelf card/* \
cardType,insertedCardType,serialNumber,\
activeFirmwareVersion,currentLP"
while { [NMSCmdRecvNextLine] } {
    set col [NMSCmdGetColumn -out 1];
    set val [NMSCmdGetColumn -out 3];
    switch -- $col {
    "cardType"{ set cardType $val; }
    "insertedCardType"{ set insertedCardType $val; }
    "activeFirmwareVersion"{ set \
    activeFirmwareVersion $val; }
    "serialNumber"{ set serialNumber $val; }
```

```
    "currentLP"{ set currentLP $val; }
"Shelf" {
    # This is the name of a card (either the first or
    # another one in the list).
    if { [string length $cardType] > 0        \
        && (   $cardType         != "none"   \
             || $insertedCardType != "none" ) } {
        # Print the info on the preceding card.
        puts [format "%-12s %-4s %-12s %-12s %-14s \
            %-14s %s" $mod $card $cardType \
            $insertedCardType $serialNumber \
            $activeFirmwareVersion $currentLP];
        set cardType "";
    }

            # Extract the card number from the name and
            # stop if maximum.
            NMSEPIPatternMatch "Card/" \
            [NMSCmdGetColumn -out 2] ""
            set card "$NMSEPI_OUTPUT_MATCH"

            if { $card > $numberOfSlots } {
                break;
            }
        }
    }
}

# Print the last card's information if required.
if { [string length $cardType] > 0        \
```

```
     && (   $cardType         != “none”   \
         || $insertedCardType != “none” ) } {
puts [format “%-12s %-4s %-12s %-12s %-14s %-14s %s\n”\
         $mod $card $cardType $insertedCardType \
         $serialNumber $activeFirmwareVersion \
         $currentLP];
}

exit 0;
```

Passport DLCI Configuration Report sample

The following Tcl script uses the NRS interface to produce a simple report for all configured Frame Relay DLCIs in the network and their major quality-of-service parameters. This example is available in the MDM load in `/opt/MagellanNMS/cfg/macros/nms/src/DlciScan.tcl`.

```
#!/opt/MagellanContrib/bin/tclsh
# Load the NMS EPI extensions for reporting
source /opt/MagellanNMS/lib/nmsepir.tcl
source /opt/MagellanNMS/lib/nmsepi.tcl

# Initialize an NRS interface.
NMSNRSInit

# Initiate a query for the various FrameRelay Service
# Parameter components.
NMSNRSStartQuery "ppc/FrameRelayUni" \
                 "ppc/EM-FrUni-Dlci-Sp"

# Report on all latest configurations (file system
# date) for all nodes.
```

```
NMSNRSAddSource -latest "ppc" ".*"


# Scan the NRS database for all matching components
while { [ NMSNRSFetchNextComponent -var comp
                                   -byname ] } {
    # Extract the major fields by name (the associative
    # array has them by type)
    if { $NMSEPI_NRS_COMPABBREV == "FrUni" } {
         set cid $comp(customerIdentifier)
         continue
    } else {
         set cir $comp(committedInformationRate)
         set bc  $comp(committedBurstSize)
         set be  $comp(excessBurstSize)
    }
    # Print the component information in
    # comma-separated format for the target system
    NMSEPIConvertCompId -disp "$NMSEPI_NRS_COMPID"
    puts "$NMSEPI_COMP_ID,$cid,$cir,$bc,$be"
}
```

Message handling

Asynchronous (i.e., event driven or event loop based) message handling in Tk is not supported (the implicit Tk event loop is incompatible with `NMSEPIEventLoop`). When using Tk, you must therefore ensure that sufficient attention (in terms of receiving replies) is given to the EPI interfaces, but not so as to affect the graphical operations (that is, without blocking). This can be achieved with the Tcl/Tk `after` command. For example, once a query is sent, `after` is used to execute a segment of code (usually a *proc* invocation) after a certain amount of time is elapsed (the rest of the code continues to execute). This called-back code must receive and

handle the next reply, if any, and at the end, re-schedule itself for the next reply. The `Recv` call should be invoked with a `0` second timeout (no-wait polling).

Example

```
proc getNextReply{} {

    while { [ NMSGMDRAPIRecvAlarm 0 ] } {
        # ... handle the received alarm...
    }

    after 1000 getNextReply; # in milliseconds

}

# ...send query or create sieve for alarms...

after 1000 getNextReply;
```

This technique leaves enough CPU time in the Tk event loop for Tk to process all of the required graphical events.

Note: The `getnextReply` function should maintain a count to limit the number of consecutive calls to the `NMSGMDRAPIRecvAlarm` command. This will avoid CPU starvation of the rest of the Tk script (in case alarms are always available to be received).

Packaging

The Tcl scripting language and several of its specializations (for example, Expect) are provided in the optional Magellan Contrib Software package, which is included in the Preside Multiservice Data Manager (MDM) CD-ROM. See the latest MDM Release Supplement for the list of available systems and their versions.

Note: These systems are not supported by Nortel Networks.

For information on installation of the Magellan Contrib Software package, see 241-6001-100 *Preside MDM Installer Guide* or 241-6001-105 *Preside MDM Software Packaging Reference Guide*.

Users are also free to install their own Tcl system, provided it is compatible with the versions indicated in the MDM Release Supplement.

Chapter 4
Perl Embedded Programming Interface

This section describes the Embedded Programming Interface (EPI) in the Perl scripting language. This chapter contains the following:

Code conventions

There are three code conventions used in this chapter:

- \
  A back slash (\) indicates that the line of code is continued on the next line space.

- #
  A message line that starts with an octothorp (#) is treated as a comment.
- =>
  when the arrow symbol (`=>`) is used to separate arguments of an EPI call, it is equivalent to a simple coma (`,`). The arrow is used only to stress that the two values are related to one another, that is, an option and its value. Option flags are sometimes quoted or not, only the one character flags (`-u`, `-h` and `-g`) truly have to be quoted.

Integration methodology

Perl Interpreter Shells can be extended by adding new built-in commands. To add new built-in commands to the Perl, the only requirement is to define them along with a special initialization routine in a dynamic shared library which is itself loaded through an appropriate Perl Module. The Perl EPI extensions are divided into two sets. To load the base, API, Command, Customer Database, and Miscelaneous interfaces of EPI, use the following two calls, or their equivalent (Perl 5.0003 or greater):

```
use lib "/opt/MagellanNMS/lib";
use EPI qw(:all);
```

where:

`use lib "/opt/MagellanNMS/lib";` adds the Preside Multiservice Data Manager (MDM) library directory to the module lookup path

`use EPI qw(:all);` loads the EPI module (which in turn loads the necessary shared libraries).

Note that the `qw(:all)` specification is optional. Use this specification to include all symbols (function and variable names) defined by the EPI module in the current naming scope. In its absence, you need to prefix all EPI symbols with `EPI::` (it is assumed in all the examples that follow that all EPI symbols have been imported).

To include the RTAC and NRS interfaces of EPI, use the following line in replacement or addition to the second line above:

```
use EPIR qw(:all);
```

The Perl EPI extension library (`libPerlEPI.so`) contains all of the required implementation functions for the base interfaces. It also contains references to all of the other Preside Multiservice Data Manager (MDM) libraries it depends on (for example, `libIPI.so`, `libAPI.so`, `libCom.so, and libipc.so`). The `libPerlEPIR.so` library contains the extensions fo the RTAC and NRS extensions.

The EPI functions have been integrated into the Perl EPI module in a very similar way as for the other supported languages. Each function call usually supports a variable number of arguments. Option flags, for example, `-iname`, are used to distinguish these various arguments. Be wary of Perl's quoting and argument list syntax. All strings must be quoted properly and each argument must be separated by a coma (`,`) or an arrow (`=>`), the later is used to clearly indicate the value of the option. Only option flags that consist of one letter (`-u`, `-h` and `-g`) need be quoted though.

Note: Perl EPI requires Perl version 5.0003 or later.

Interface

Perl EPI provides five groups of built-in commands:

- **Base**:
  Provides mapping to the EPI base routines and utilities as well as additional housekeeping routines.
- **Generic API Access**:
  Provides access to the Generic Application Programming Interface (API) commands and replies.
- **Specialized API Access**:
  Provides specialized and value-added access to specific API Providers such as the Alarm and Status, Network Model, and Host Group Directory Service (HGDS) APIs.
- **Command Access**:
  Provides access to the Preside Multiservice Data Manager (MDM) command macro capabilities.
- **Customer Database Access:**
  Provides access to the MDM Customer Database capabilities.

- **Real-Time Alarm Collection (RTAC)**
  Provides access to the spooled alarms collected by RTAC.
- **Network Reportinng System (NRS)**
  Provides access to the network-wide configutation data collected by NRS.

Command usage information

The following information applies to built-in commands:

- **help**
  All commands support a -h (help) option that displays arguments to a command
- **command argument**
  You need to specify all command arguments in the indicated order.
- **unnamed and named connections**
  By default, the connection-oriented commands (API or Command Access) operate on a single unnamed connection. It is possible, however, to create concurrent named connections. To do so, specify the connection name as an argument (`-iname`) to a command (except for the `NMSEPIDefaultAPI`, `NMSEPIDefaultCmd`, `NMSEPIDefaultCdb`, `NMSEPIDefaultRTAC`, or `NMSEPIDefaultNRS` commands).
- **error output**
  Error messages are usually output to the standard error stream. You can stop output to the standard error stream by using the NMSEPISet command.
- **return codes**
  The commands usually return `1` upon success and `0` upon failure. The global variable `NMSEPI_RESULT` stores the reason for failure. The possible values and the corresponding meanings for `NMSEPI_RESULT` are as follows:

0	success
1	error in argument list
2	default/named interface not created/initialized
3	default/named interface not connected to server
4	operation failed

5 operation timed out

6 no more replies

7 attempt to create an exiting interface (name)

This differs from DtKsh EPI, where commands return the error code. This difference allows you to use EPI commands in Perl flow control structures, where `1` indicates success (in DtKsh and most other Shells, `0` indicates success).

Example

```
if ( NMSGMDRAPIConnect(-ssel) ) {
... # successfully connected
}
```

Some commands also support an `-out` argument. This argument forces EPI to return its (string) result instead of the return codes.

Base

In general, the first task in writing Perl EPI scripts is to initialize an interface. There are specific API and Command Access commands to do this, but the Base also provides a general routine:

- **NMSEPIInit()**
  Initializes the EPI environment.
- **NMSEPITerm()**
  Drops all current API and command interfaces and terminates the IPC environment. This is sometimes required when another EPI script (which makes and maintains its own API or command interfaces) is reused and invoked in the same UNIX process. The second script would otherwise fail because of duplicate server connections.

EPI normally outputs error messages (for example, if the connection to a server fails to be created) on the standard error stream. This can be prevented with the following command:

- **NMSEPISet (+err|-err)**
  Controls whether error messages are output (`+err`) or not output (`-err`).

API and Command Access commands usually apply to a default connection (a single connection for all APIs, another one for Command Access, and another one for Customer Database access). When using multiple concurrent connections, you can use the following commands to make a specific command the default and thereby avoid having to specify its name as an argument in every command:

- **NMSEPIDefaultAPI([-iname=><name>])**
  NMSEPIDefaultCmd([-iname=><name>])
  NMSEPIDefaultCdb([-iname=><name>])
  If it exists, makes the named interface connection the default. If no interface is named, the original unnamed default interface, if any, is set to be the default.

  Note: There is only one default API interface for all API types (for example, Alarm and Status, Network Model).

The following MDM utilities are supported:

- **NMSEPIConvertCompId ([-out,]**
  [-canon|-disp|-type|-mnem|-ep1|
  -ep2|-dpn|-switch,] <component ID>)
  Converts the specified component ID and returns the result in the `NMSEPI_COMP_ID` module variable and as a string result (if `-out` is used). The conversions are as follows:

  `-canon`
  converts to canonical API format (for example, `PM AM1 PE 1 PI 1`).

  `-disp`
  converts to display format (for example, `PM/AM1 PE/1 PI/1`).

  `-type`
  extracts the module/link type (for example, `PM` or `NL`).

  `-mnem`
  extracts the module mnemonic (for example, `AM1`).

  `-ep1` and **`-ep2`**
  extract the first and second link endpoints in canonical API format.

-dpn
converts a DPN-100 OA or a PE/PI/PO component ID to a form suitable for commands (<mnemonic> [pe <pe#>|<pi #> [<po #>]]). (For example, AM1 11.)

-switch
returns the module-level component ID in canonical API format (for example, PM AM1).
-omni
returns the Preside Multiservice Data Manager (MDM) HP-OpenView DeskTop compatible component ID (similar to display format except for link names).

Examples

```
NMSEPIConvertCompId(-ep1, "$myLinkId");
print "Endpoint1: $NMSEPI_COMP_ID\n";

NMSEPIConvertCompId(-dpn, "$myDPNPort");
NMSCmdSendCommand($myOA,"$NMSEPI_COMP_ID enable");
```

- **NMSEPIConvertTime([-out,] [-stc,]**
  -api|-tostc|-epoch|-unix|-alarm|-pp
  |-ftime=><format>
  |-offset=>"<nb> [days]"
  [, <time string>])

  Converts the specified time string and returns the result in the NMSEPI_OUTPUT_TIME global variable and as a string result (if -out is used). The input time can be in:

  API format (YYYY MM DD HH MM SS)
  Common Alarm format (YY-MM-DD HH:MM:SS)
  UNIX Epoch (<number of seconds since 1970>)
  Passport reply format (YYYY-MM-DD HH:MM_SS)

  The returned time is in the same time frame as the input one. However, exceptions occur if you specify -stc or if you select the -tostc conversion. If you specify -stc the input time is assumed to be in Standard Time Coordinates, that is, Greenwich Mean Time (GMT) . If

you use `-stc` and not `-tostc`, then the output time converts from STC to local workstation time. If you do not specify the input time, the current workstation time is used and `-stc` is ignored.

The conversions are as follows:

`-api`
returns the time in API format.

`-tostc`
produces the time in API format converted to Standard Time Coordinates.

`-epoch`
returns the time as a UNIX Epoch value (number of seconds since 1970).

`-unix`
returns the timeas the default time format for the workstation's LOCALE (see `man -s3c ctime`)

`-alarm`
returns the time in Common Alarm format.

`-pp`
returns the time in Passport reply format

`-ftime=><format>`
returns the time in the specified format (see `man -s3c strftime`, 100 characters maximum).

`-offset=><nb> [, days]`
if `days` are not specified, returns the time in API format after applying the specified positive or negative offset in seconds, otherwise returns days

`-secsinday`
returns the number of seconds from the previous midnight and the specified time (or the current time)

`-stcoffset`
returns the number of seconds between the local workstation time and the

coordinated universal time (UTC) and, if needed, taking daylight savings time into consideration,. The offset is positive going west from UTC (same as UNIX timezone/altzone).

Example

```
NMSEPIConvertTime(-offset=>"-7", days);
print "$NMSEPI_OUTPUT_TIME\n";
```

returns the time 7 days ago, for example:

```
2000 03 24 17 01 19
```

```
NMSEPIConvertTime(-epoch, "2000 03 24 17 01 19");
print "$NMSEPI_OUTPUT_TIME\n";
```

returns the same as a UNIX epoch, for example:

```
953935279
```

```
NMSEPIConvertTime(-ftime=>
                  "Time was: %a %b %d %l:%M%p", \
                  "953935279");
print "$NMSEPI_OUTPUT_TIME\n";
```

returns the same information using a custom format:

```
Time was: Fri Mar 24  5:01PM
```

- **NMSEPICompareCompIds(<component ID1>, <component ID2>)** Compares the two returning component IDs and returns (instead of the usual return codes) `0` if they are identical, `<0` if the first component ID precedes the second, and `>0` if the second precedes the first.
- **NMSEPIPatternMatch ([-out,] ["-g",] <pattern>, <target> [, <substitute>])**

  Performs pattern matching of <pattern> with <target>. Specify <pattern> with grep syntax. You can include one \c\ delimited sub-pattern, which is returned if there is a match instead of the full matched portion. If you specify <substitute>, the matching sub-string is replaced by <substitute>. The `-g` option substitutes all matching sub-strings. The matched or substituted string is returned in the `NMSEPI_OUTPUT_MATCH` module variable (and returned as a string if `-out` is used).

Example

```
if ( NMSEPIPatternMatch("pe \([0-9]*\)down",
                "$NMSEPI_OUTPUT_TEXT") ) {
  print "PE# $NMSEPI_OUTPUT_MATCH is down.\n";
...
```

- **NMSEPIGetContext([-out,] [-ssel|-user|-ws,] <var. name>)**
  NMSEPISetContext(-user|-ws, <var. name>, <value>)
  Gets or sets a Preside Multiservice Data Manager (MDM) context variable. The context corresponds to the Service Selection for `-ssel`, the current User Session for `-user`, and the Workstation Wide for `-ws`. `NMSEPIGetContext` sets the `NMSEPI_CTX_VALUE` module variable to the extracted value (and returned as a string if `-out` is used).

  Example

  ```
  NMSEPIGetContext(-user, "DPN_QUICK_STEP");
  print "Hot context: $NMSEPI_CTX_VALUE\n";
  ```

- **NMSEPIRegisterContextInterest([-user|-ws**
  <callback string> ,
  <var. names...>])
  Binds the specified Perl command string, `<callback>`, (typically a function invocation and its arguments) to the specified Preside Multiservice Data Manager (MDM) Context variables and domain (`-user` for the User Session and `-ws` for Workstation Wide). This command string will be executed whenever one of the named variable changes value and the script is in an event loop (`NMSEPIEventLoop`). The following variables are available to the callback:

  `NMSEPI_CB_REASON`
  is set to `CONTEXT`.

  `NMSEPI_CTX_VAR`
  is set to the the named of the changed variable.

  `NMSEPI_CTX_VALUE`
  is set to the new value for the variable.

  Only one such callback can be registered. To deregister the callback, call this command without any arguments.

Example

```
sub componentHotContext {
    # react to the hot context selection
    # whose component name value is in
    # $NMSEPI_CTX_VALUE
    ...
}
...
NMSEPIRegisterContextInterest(-user,
    "componentHotContext", "DPN_QUICK_STEP");
...
NMSEPIEventLoop();
```

- **NMSEPIEventLoop()**
  Initiates an Xt-based event loop for asynchronous message handling. This command never returns. The processing scripts are then performed by the callback Perl code that is bound to the API and command interfaces (see `NMSEPIRegisterContextInterest`, `NMSAPIBindCallback`, `NMSCmdBindCallback`, and `NMSCdbBindCallback`)

- **NMSEPITimer(set, <msec>, <callback>)**
  NMSEPITimer(clear, <timer name>)
  The first form, `set`, of this command binds the specified Perl command string, `<callback>`, (typically a function invocation and its arguments) to be invoked once in `<msec>` milli-seconds when the script is in an event loop (`NMSEPIEventLoop` or `XtMainLoop`). The created timer is given a name returned as the value of the `NMSEPI_TIMER_NAME` environment variable. This name can be used in the second form, `clear`, to cancel the timer. The callback code is invoked with the following variables:

  `NMSEPI_CB_REASON`
  is set to `TIMER`.

  `NMSEPI_TIMER_NAME`
  is set to the name of the timer that has expired.

  Example

```
sub rescanList {
    ...
    # re-create the timer to be called again
```

```
    NMSEPITimer(set, (5 * 60 * 1000), "rescanList");
}
...
NMSEPITimer(set, (5 * 60 * 1000), "rescanList");
...
NMSEPIEventLoop();
```

- **NMSEPILongArith ([-out,] ["-u",] <integer value>, "+"|"-"|"x"|"/"|"%", <integer value>)**

  Performs long integer arithmetic on the values provided (`+` addition, `-` subtraction, `x` or `*` multiplication, `/` division, and `%` modulo). If `-u` is used, unsigned arithmetic is performed. The results are returned as a string in the `NMSEPI_RESULT` module variable and as a string result if `-out` is specified.

  Example

```
NMSEPILongArith($currByteCount, "-",
                $prevByteCount);
print "Delta: $NMSEPI_RESULT\n";
```

Generic API access

These commands provide access to the Generic API at the same level as if you were using the Generic API Provider utility, *genapi*, directly. At this level, no special knowledge of the individual API types is required; all queries can be handled in a generic way. (For more information, see 241-6001-200 *Preside MDM Application Programming Interface Primer*.

Generic API Access commands are used in the following sequence:

1 Initialize an API interface.

2 Connect to the API server.

3 If the server requires it, send a `REGISTER` message and wait for its reply.

4 Send an API query.

5 Receive the next reply and extract the needed information.

6 Disconnect from the API server.

7 Drop the API interface.

Multiple queries can be issued, but individual API interfaces will handle these queries synchronously, one at a time (except for sieve event notifications). It is possible, however, to create multiple-named API interfaces (each one resulting in its own connection to the server and therefore capable of independent, parallel queries). Note also that a single API interface can be reused (disconnected and reconnected to another server).

The same command sequence is used by the Specific API Access areas, but specialized commands are sometimes added to handle the particular details of an API type. For more information on specialized commands, see “Specialized API access” (page 263).

Generic API Access commands

The following Generic API Access commands are provided:

- **NMSAPIInit([-iname=><name>,] <API dictionary path>)**
  Initializes a Generic API interface using the indicated API dictionary. If more than one interface is to be used, you should provide a name for it. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

  Note: Specialized API Access provides its own version of this command, which hides the detail of the API dictionary path. For more information, see “Specialized API access” (page 263).

- **NMSAPIDrop(-iname=><name>)**
  Drops the named API interface to clear up the connection or so it can be reused.

- **NMSAPIConnect([-iname=><name>,] <service name> [, <host>])**
  Connects the default or named interface to the specified API server, optionally on a different Preside Multiservice Data Manager (MDM) host.

  Example

```
NMSAPIInit(“/opt/MagellanNMS/lib/api/GMDR.dict”);
if( NMSAPIConnect(“GMDR”, “localhost”) ) {
    ... # connection successful
```

- **NMSAPIDisconnect([-iname=><name>])**
  Disconnects the default or named interface from its current API server.

- **NMSAPIRegister([-iname=><name>,] <user id> [, <password>] [, -attr=><attribute lines>])**

  Sends a `REGISTER` message through the default or named interface with the specified parameters

  where:

  `-attr=><API attribute lines>`
  is one or more API attribute lines to add to the message (for example, the `"_attr: userCapability E mdInject"` line needed to register to IMDR with alarm injection capabilities).

  This command is actually a combination call, since it will both send the query and wait for the reply. The return code therefore indicates the success or failure of the complete `REGISTER` command-reply sequence.

  Example

  ```
  if ( NMSAPIRegister("toto",
       -attr=>"_attr: userCapability E mdInject") {
  ... # successful register for alarm injection
  ```

- **NMSAPISendCommand([-iname=><name>,] <API query string>)**

  Sends a query to the default or named interface. The query is specified as an ASCII string in API syntax.

  Example

  ```
  NMSAPISendCommand("_cmd: get
  _obj_type: node
  _obj_id: compId NI PM R78
  _attr_id: all
  ");
  ... # receive replies
  ```

- **NMSAPIRecvReply([-iname=><name>,] [-out,] [-var=><array name>,] [<timeout>])**

  Waits for and receives the next reply record from the server. A timeout

can be specified (by default, it waits forever). With a timeout of `0`, the command acts as a no-wait poll. If `-out` is specified, the received record is returned as a string in an API-like format instead of the exit code.

If a variable name is specified with `-var`, the named variable is set as a hash array containing the received API fields. For API fields having an attribute name, the array key is set to:

```
<API label (without the :)>,<API attribute name>
```

and for the others, the key is set to:

```
<API label (without the :)>
```

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

Example

Receving a GET command reply for a Network Model node could lead to the following values:

```
while ( NMSAPIRecvReply(-var=>reply) ) {
   #...
   # $reply{_obj_type} is “node”
   # $reply{_obj_id,compId} is “PM R78”
   # $reply{_attr,rawState} is “INSV”
   #...and so on...
```

It is possible to list all the array keys using Perl’s `“keys %<array name>”` construct.

The return error code is in `NMSEPI_RESULT` (`5` indicates a timeout occurred).

- **NMSAPISkipRestOfReply([-iname=><name>])**
  This combination call waits for and ignores all further replies until the end of response. If it finds an error record, the `NMSEPI_RECORD_TYPE` module variable is set to `ERROR`; otherwise it is set to `ENDRESP`.

- **NMSAPIGetRecord([-iname=><name>,] [-out,] [-var=><array name>])**
  Extracts the last received record’s (`NMSAPIRecvReply`) record type (`NONE`, `REGISTER`, `RESPONSE`, `EVENT`, `ENDRESP`, `ERROR`, or `END`). It places this information in the `NMSEPI_RECORD_TYPE` module variable (and returned it as a string if `-out` is used) and resets the API field list

to the beginning for `NMSAPIGetNextField`, `NMSAPIFindNextField`, and `NMSAPIFindNextAttr` (it can therefore be called several times to repeatedly scan the field/attribute list).

If a variable name is specified with `-var`, the named variable is set as a hash array containing the received API fields. For API fields having an attribute name, the array key is set to:

```
<API label (without the :)>,<API attribute name>
```

and for the others, the key is set to:

```
<API label (without the :)>
```

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

It is possible to list all the array keys using Perl's "`keys %<array name>`" construct.

Example

```
while ( NMSAPIRecvReply() ){
    NMSAPIGetRecord();
    if ( $NMSEPI_RECORD_TYPE eq "ENDRESP" ) {
         ... # got end of response
```

- **NMSAPIGetNextField([-iname=><name>,] [-out])**
  Extracts the next API field from the last received reply. The following module variables are set:

  `NMSEPI_FIELD_LABEL`
  is the API field type (`_attr:`, `_end_resp:`, `_error:`, `_end:`, `_obj_type:`, `_obj_id:`, `_user_id:`, `_capability:`, `_inv_id:`, `_event_type:`, `_time:`, or `_sieve_id:`).

  `NMSEPI_FIELD_NAME`
  is the API field name element (`_ATTR:` and `_OBJ_ID:` only).

  `NMSEPI_FIELD_TYPE`
  is the API field type element (`_ATTR` and `_OBJ_ID` only).

  `NMSEPI_FIELD_VALUE`
  is the API field value.

Multiple-line attribute block values are returned as a single multiple-line value.

If `-out` is used, the field is returned as a string in an API-like format.

Example

```
NMSAPIGetRecord();
...
while ( NMSAPIGetNextField() ) {
   if (   ($NMSEPI_FIELD_LABEL eq "_attr:")
       && ($NMSEPI_FIELD_NAME  eq "time") ) {
   print "Time is: $NMSEPI_FIELD_VALUE\n";
...
```

- **NMSAPIFindNextField([-iname=><name>,] [-out,] <label>)**
  Locates and returns the next API line of the type specified by `<label>` (`_attr:`, `_end_resp:`, `_error:`, `_end:`, `_obj_type:`, `_obj_id:`, `_user_id:`, `_capability:`, `_inv_id:`, `_event_type:`, `_time:`, or `_sieve_id:`) from the API field list of the last received API reply. The module variables are set as for `NMSAPIFindNextField`. If `-out` is used, the field is returned as a string in an API-like format (but without the label) on standard output. The previous example can be rewritten as follows:

```
NMSAPIGetRecord();
...
while ( NMSAPIFindNextField("_attr:") ) {
   if ( $NMSEPI_FIELD_NAME eq "time" ) {
      print "Time is: $NMSEPI_FIELD_VALUE\n";
...
```

- **NMSAPIFindNextAttr [-iname=><name>,] [-out,] <attr. name>**
  Locates and returns the next named attribute from the API field list of the last received API reply. The module variables are set as for `NMSAPIFindNextField`. If `-out` is used, the field value is returned as a string. The previous example can be rewritten as follows:

```
NMSAPIGetRecord();
...
if ( NMSAPIFindNextAttr("time") {
   print "Time is: $NMSEPI_FIELD_VALUE\n";
...
```

- **NMSAPIBindCallback([-iname=><name>,] <callback command>)**
  Binds the specified Perl command string (typically a function invocation and its arguments) to the API interface. This command string will be executed whenever a new message is received from the server for this interface and the script is in an event loop (`NMSEPIEventLoop`). The following module variables are available to the callback:

  `NMSEPI_INAME`
  is the API interface name of the callback.

  `NMSEPI_CB_REASON`
  is the reason for the callback (`LOST_CONNECTION`, `ERROR`, `EVENT`, or `RESPONSE`).

  `NMSEPI_RECORD_TYPE`
  is the returned record (API record type or `ALARM` if the received record is an Alarm, or `RAWSTATE` if it is a Raw State).

  `NMSEPI_ALARM_SEVERITY, NMSEPI_ALARM_EVENT, NMSEPI_ALARM_FAULT, NMSEPI_ALARM_TIME, NMSEPI_ALARM_COMPID, NMSEPI_ALARM_OPDATA`
  are the special alarm fields

  `NMSEPI_ALARM_TIME, NMSEPI_ALARM_COMPID, NMSEPI_RAW_STATE`
  are the alarm fields if a Raw State Change record is received

  `NMSEPI_API_SIEVEID`

is the source Sieve ID for the received message, if any

In addition, the post-`Recv` functions (i.e., `NMSAPIGetRecord`, `NMSAPIGetNextField`) can be used in the context of this callback to extract other API fields and attributes from the received reply. The event loop is launched with `NMSEPIEventLoop` and never returns.

Example

```
sub mycb {
   # ...Process the reply...
}
# ... launch commands and create sieves ...
NMSEPIBindCallback(“mycb”);
...
NMSEPIEventLoop(); # never returns
```

Specialized API access

This section describes additional commands as well as value-added versions of the Generic API Access commands that are built to interact with specific APIs and their features.

Alarm and Status API access

This area adds commands to interact with the Alarm and Status API. In addition to a number of combination calls (for example, to create an alarm sieve or to inject alarms), it also supports automatic extraction of the major alarm attributes to support most forms of network automation.

Alarm and Status API Access commands

The following Alarm and Status API Access commands are provided:

- **NMSGMDRAPIInit([-iname=><name>])**
  This simplified form of the `NMSEPIInit` command initializes an API interface to the Alarm and Status API server (GMDR). If multiple API interfaces are to be used, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).
- **NMSGMDRAPIConnect([-iname=><name>,] [-ssel | <host>])**
  Connects to the Service Selected `GMDR` server (if `-ssel` is specified), the specified location (`<host>`), or the local host (by default).

Example

```
NMSGMDRAPIInit(-iname=>"toto")
if ( NMSGMDRAPIConnect(-iname=>"toto", -ssel) ) {
              ... # we're connected
```

- **NMSGMDRAPICreateAlarmSieve([-iname=><name>,] [-all,] [-attr=><alarm event filters and attributes...>])**

  This combination call creates a simplified alarm sieve and waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the major alarm attributes (`compId`, `time`, `severity`, `event`, `faultCode`, and `operatorData`) of all of the received alarms. However, if `-all` is specified, all of the attributes are extracted. You can add event filter (`"_attr: eventFilter SS <attribute> <operator> <type> <value>"`) and specific attribute extraction (`"_attr: eventInfo S <attribute>"`) parameters in API syntax using the `-attr` option and its values (note that each value may be multiple-lined).

  Example

  ```
  if ( NMSGMDRAPICreateAlarmSieve(-attr=>
       "_attr: eventFilter SS event EQ E SET",
       "_attr: eventInfo S commentData") ) {
     ... # sieve creation successful
  ```

  The notifications from the sieve(s) can be collected with `NMSAPIRecvReply` or with `NMSGMDRAPIRecvAlarm` (with the `EVENT` record type).

  Each Alarm and Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned as the `NMSEPI_API_SIEVEID` variable.

  Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **NMSGMDRAPIRecvAlarm([-iname=><name>,] [-out,] [-var=><array name>,] [<timeout>])**

  Enhances `NMSAPIRecvReply` by waiting for and receiving the next reply and automatically extracting the major alarm fields in their own module variables (if the reply is the result of an alarm *get* or notification). The following module variables are set:

  NMSEPI_RECORD_TYPE:
  is the API record type.

  NMSEPI_ALARM_SEVERITY:
  is the severity of the alarm.

  NMSEPI_ALARM_EVENT
  is the alarm event.

  NMSEPI_ALARM_FAULT
  is the fault code.

  NMSEPI_ALARM_TIME
  is the alarm time.

  NMSEPI_ALARM_COMPID
  is the component ID.

  NMSEPI_ALARM_OPDATA
  is the operator data.

  For information on the legal values of these fields, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The other attributes, if any were specified, can be extracted with the `NMSAPIGetRecord`, `NMSAPIGetNextField`, `NMSAPIFindNextField`, and `NMSAPIFindNextAttr` commands.

  Example

```
NMSGMDRAPICreateAlarmSieve(-attr=>
   "_attr: eventFilter SS event EQ E SET");
...
if ( NMSGMDRAPIRecvAlarm() ) {
```

```
if (    ("$NMSEPI_RECORD_TYPE"    eq "EVENT")
     && ("$NMSEPI_ALARM_SEVERITY" eq "critical")){
    ... # handle the critical alarm
```

If `-out` is used, the default alarm fields are returned as a string consisting of a sequence of at least six lines (for the `severity, event, faultCode, time, compId,` and `operatorData` respectively).

If a variable name is specified with `-var`, the named variable is set as a hash array containing the received API fields. For API fields having an attribute name, the array key is set to:

```
<API label (without the :)>,<API attribute name>
```

and for the others, the key is set to:

```
<API label (without the :)>
```

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

Note: Some lines may be empty if the corresponding field is not present. As well, `operatorData` can consist of more than one line.

- **NMSGMDRAPIFormatAlarm([-iname=><name>,] [-out,] [terse|normal|full])**

  This command produces the last received alarm (from `NMSAPIRecvReply`, `NMSGMDRAPIRecvAlarm` or from a callback) in the Preside Multiservice Data Manager (MDM) Common Alarm Format (as used in the Alarm Display and Component Information Viewer tools). The alarm can be formatted in either `terse` (one line), `normal` (includes Operator Data) or `full` (all information) formats, the default is `full`. The `NMSEPI_ALARM_DISPLAY` variable is set to the resulting string.

  If `-out` is used, the resulting string is also returned.

- **NMSGMDRAPIInjectAlarm([-iname=><name>,] <comp ID>, <event>, <severity>, <fault code>, <notification ID>, <comment> [, -time=><time>] [, -attr=><other attributes...>])**

  Sends an alarm injection command with the following specified parameters:

`comp ID`
is the component ID.

`event`
is the alarm event (`set|clear|message`).

`severity`
is the severity of the alarm (`warning|minor|major|critical|cleared|indeterminate`).

`fault code`
is the fault code in `AAAACCCC` format.

`notification ID`
is the sequence number (if 0 or an empty string, a unique value will be provided).

`comment`
is the comment data string.

As well, a time can be specified using `-time` (in *`yyyy mm dd hh mm ss`* format) and other attributes specified using `-attr` and its values (as one or more "`_attr: <name> <type> <value>`" per string). Unspecified attributes take default values (see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*).

Note: This is not a combination call since the command does not wait for the server's reply, which must then be explicitly received or ignored.

Example

```
NMSGMDRAPIInjectAlarm("MYRTR WEST3","set"
      "major", "A0000001", 23,
      "The router does not reply.",
      -attr=>"_block: _attr operatorData S
Try: 12, Timeout: 5, IP: 33.24.1.1
_end_block:");
NMSAPISkipRestOfReply();
```

- **NMSGMDRAPICreateRawStateSieve([-iname=><name>,] [-attr=><event filters...>])**

  This combination call creates a simplified raw state change sieve and

waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the `compId`, `time`, and `rawState` attributes. It is possible to add an event filter (`"_attr: eventFilter SS <attribute> <operator> <type> <value>"`) using the `-attr` option and its values (note that each value may be multi-lined).

Example

```
if ( NMSGMDRAPICreateRawStateSieve(-attr=>
    "_attr: eventFilter SS rawState EQ E OOS" ) ) {
    ... # sieve creation successful
```

The notifications from the sieve(s) can be collected with `NMSAPIRecvReply` or with `NMSGMDRAPIRecvRawState` (using the `EVENT` record type).

Each Alarm and Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned as the `NMSEPI_API_SIEVEID` variable.

Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **NMSGMDRAPIRecvRawState([-iname=><name>,] [-out,] [-var=><array name>,] [<timeout>])**

  Enhances `NMSAPIRecvReply` by waiting for and receiving the next reply and automatically extracting the raw state change fields in their own module variables (if the reply is the result of a node *get* or notification). The following module variables are set:

  `NMSEPI_RECORD_TYPE`
  is the API record type.

  `NMSEPI_ALARM_TIME`
  is the notification time.

NMSEPI_ALARM_COMPID
is its component ID.

NMSEPI_RAW_STATE
is its raw state value.

For information on the legal values of these fields, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*.

Example:

```
NMSGMDRAPICreateRawStateSieve(-attr=>
   "_attr: eventFilter SS rawState EQ E OOS");
...
if ( NMSGMDRAPIRecvRawState() ) {
   if ( "$NMSEPI_RECORD_TYPE" eq "EVENT" ) {
      ... # handle the out-of-service component
```

If `-out` is used, the raw state fields are returned as a string consisting of a sequence of three lines (for the `rawState, time,` and `compId` respectively).

If a variable name is specified with `-var`, the named variable is set as a hash array containing the received API fields. For API fields having an attribute name, the array key is set to:

`<API label (without the :)>,<API attribute name>`

and for the others, the key is set to:

`<API label (without the :)>`

If an API attribute has multiple values in the reply, these values are concatenated in the array value with carriage returns as separators.

Network Model API access

The Network Model (NM) API access allows you to initialize an API interface to the server and connect the interface to the host.

Network Model API Access commands

Network Model (NM) API Access provides two additional commands:

- **NMSNMAPIInit([-iname=><name>])**
  Initializes an API interface to the NM Server. If multiple API interfaces are to be used, specify a name in the command invocations for this

interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

- **NMSNMAPIConnect([-iname=><name>,] [-ssel | <host>,])**
  Connects the interface to the Service Selected NM Server host (`-ssel`), the named host, or the local host (by default).

  Example

```
NMSNMAPIInit();
if ( NMSNMAPIConnect("bcars561") ) {
    ... # we're connected
```

Various Generic API Access commands can be used to send queries and receive their replies.

Host Group Directory Service API access

The Host Group Directory Service (HGDS) API Access allows you to extract data on the Passport Group configuration of the Preside Multiservice Data Manager (MDM). Some value-added and combination calls are added to the Generic API Provider along with automatic extraction of the Passport Member information.

HGDS API Access commands

The following HGDS API Access commands are provided:

- **NMSHGDSAPIInit([-iname=><name>])**
  Initializes an API interface to the Host Group Directory Server. If multiple API interfaces are to be used, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultAPI` to make it the default interface).

- **NMSHGDSAPIConnect([-iname=><name>,] [-ssel|<host>])**
  the named host, or the local host (by default).

  Example

```
NMSHGDSAPIInit(-iname=>"hgds");
if ( NMSHGDSAPIConnect(-iname=>"hgds", -ssel) ) {
    ... # we're connected
```

- **NMSHGDSAPISendQuery([-iname=><name>,]**
  -group [=><group name>]
  | -member [=><member name>]
  | -child=><group name>
  | -parent=><member name>)

  Sends an HGDS API query. Use `-group` to retrieve the named Passport Group (or all of them if no name is indicated), `-member` for the named Passport module (or all of them if no name is indicated), `-child` for the Passport hosts in the named group, or `-parent` for the groups containing the named host.

- **NMSHGDSAPIRecvReply([-iname=><name>,] [-out,] [<timeout>])**
  Enhances `NMSAPIRecvReply` to automatically extract the Passport host information if present in the module variables. The following module variables are set:

  NMSEPI_RECORD_TYPE
  is the reply record type.

  NMSEPI_HGDS_NAME
  is the Passport host or group name.

  NMSEPI_HGDS_IPADDR
  is the Passport host IP address, if applicable.

  Example

```
if ( NMSHGDSAPISendQuery(-iname=>"hgds",
              -child=>"$grp" ) ) {
  while ( NMSHGDSAPIRecvReply(-iname=>"hgds") ) {
      if (system("ping $NMSEPI_HGDS_IPADDR") == 0){
         ... # Passport is reachable
```

Command access

Like the `cmccmd` utility, Command Access allows scripts to connect to Passport Groups and DPN-100 OAs (the command route), send commands to the modules they contain, and receive the replies. Command Access also provides several utility commands to assist with the parsing and identification of the command output. Command Access commands further communicate with the Command Session servers (CMCFUN and CM) that correspond to the current `DISPLAY` environment variable (for example, for a script that uses

the session servers in the current Preside Multiservice Data Manager (MDM) User Session and potentially uses its current group and OA connections) in the Command Console.

For stand-alone (for example, CRON) or specific macros (for example, using Passport provisioning mode), it may be necessary to create a private Command Session for the execution of the script. Command Access provides the `NMSCmdSession` comand to start (and stop) a session (For a description of `cmcwrap` and how to use it to write macros, see 241-6001-301 *Preside MDM Customization Administrator Guide*).

Command Access commands are used in the following sequence:

1 Initialize a command interface.

2 Connect to the Command Session server (CMCFUN).

3 Connect to one or more Passport Group(s) or OA(s), if necessary.

4 Send a command to a node in the connected Groups and OAs.

5 Receive the command replies, either as a single string or one line at a time.

6 Use Perl or the provided commands to analyze the reply.

7 Disconnect from the command server.

8 Drop the command interface.

Multiple commands can be sent to the nodes, but individual command interfaces will handle these commands synchronously, one at a time. It is possible, however, to create multiple named command interfaces (each one resulting in its own connection to the server and therefore capable of independent, parallel commands).

Command Access commands

The following Command Access commands are provided:

- **NMSCmdSession([-display=><DISPLAY>,] start|stop [, -dpn=><MDM host>] [,-pp=><MDM host>] [,-idle=><timeout>])**

  Starts (`start`) or stops (`stop`) a private command session for the

specified <DISPLAY> name (must be a unique value workstation-wide for private sessions). This is equivalent to spawning the following command in the background:

```
/bin/env DISPLAY=<DISPLAY> \
   /opt/MagellanNMS/bin/loop
          -delay 3 \
               /opt/MagellanNMS/bin/icm \; \
               /opt/MagellanNMS/bin/cmcfun
```

Since `loop` is used, the session terminates automatically within 30 seconds if the script that has spawned it terminates without stopping it (be careful of middle shell processes). See 241-6001-301 *Preside MDM Customization Administrator Guide* for more information on the `loop` utility.

The calling script waits for two to three seconds after invoking the command (or performs whatever operations it wants to the same effect) before attempting to connect to the session in order to let the session servers initialize themselves properly.

The calling script waits for two to three seconds after invoking this command (or performs whatever operations it wants to) before attempting to connect to the session to let the session servers initialize themselves properly.

If one of `-dpn and/or -pp` is specified, the matching service selection is applied to the new session so that the corresponding servers are used instead of the current workstation service selection. See `NMSCmdSetServiceSelection` to change the session's service selection.

If `-idle` is specified with a non-zero value, the time (in minutes) is passed to cmcfun which self-terminates automatically if the specified time elapses with no command activity. This ensures that device connection resources are freed if not used for an extended period of time. The next time the session is used, the required group connection needs to be re-created.

Example

```
# start a private command session and wait for it
# to initialize
if (NMSCmdSession(-display=>$mysession) != 1 ) {
    exit(1);
}
sleep(3);
# initialize an connect the command interface
# with -display $mysession (see below)
```

- **NMSCmdInit([-iname=><name>])**
  Initializes a command interface. If multiple interfaces are to be used at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultCmd` to make it the default interface).

- **NMSCmdDrop(-iname=><name>)**
  Drops the named command interface (frees allocated resources).

- **NMSCmdConnect([-iname=><name>,] [-display=><DISPLAY>])**
  Connects the interface to the current Command Session servers (for interactive scripts used within an Preside Multiservice Data Manager (MDM) user session or as argument to `cmcwrap`) or the servers that correspond to the specified `DISPLAY` variable for a previously started alternate Command Session server set.

  Example

  ```
  NMSCmdInit();
  if ( NMSCmdConnect(-display=>$mysession) ) {
  ... # we're connected
  ```

- **NMSCmdDisconnect([-iname=><name>])**
  Disconnects the interface from the session servers. The interface may be reconnected to the servers from the same session or to servers from another session.

- **NMSCmdSetServiceSelection([-iname=><name>,]**
  [-dpn=><MDM hostname>,]
  [-pp=><MDM hostname>])
  Controls the Service Selection settings for the Command Session this interface is connected to. `-dpn` specifies the MDM host name for DPN Network Access. `-pp` specifies the MDM host name for Passport

Network Access.
This is similar to using the Service Selection tool on the same session as the Session Servers. If other users (or programs/scripts) are using that session, then they will also be impacted by this change. The Command Interface must be connected, see `NMSCmdConnect`) for this function to work.

- **NMSCmdSendConnect([-iname=><name>,] OA|GROUP, <route>, <name>, <password>)**
  Sends a connect request to the indicated route using the specified authentication information. If an error is found, the error text is the value of the `NMSEPI_OUTPUT_TEXT` module variable.

  Note: Connecting to an already connected route results in an error, even though the route is available. Send a `""` command to the route to verify if a connection has already been established.

  Example
  ```
  NMSCmdSendCommand(myOA, "");
  if (   NMSCmdSkipRestOfReply()
      || NMSCmdSendConnect(OA,
                "myOA", "myCap", "aq1sw2") ) {
     ... # we're connected to the route
  ```

  Note: If the script is to be invoked from the Command Console, you can use `"$ENV{'CMC_CURRENT_ROUTE'}"` (with the quotes as indicated) here and in `NMSCmdSendCommand` to indicate the current Command Console route.

- **NMSCmdSendDisconnect([-iname=><name>,] <route>)**
  Disconnects the interface from the named route.

- **NMSCmdSendCommand ([-iname=><name>,] <route>, <command>)**
  Sends a command to a node through the specified connected route. The name of the destination node is typically the first token of the command.

  Example
  ```
  if ( NMSCmdSendCommand("myOA", "R78 d") ) {
     ... # command sent
  ```

Note: The special command routes of the Command Console ($ for macro access, @ for the SNMP Command Framework, and * for the Passport wild-card group) can all be used as routes from EPI though, obviously, there is no need to first connect to them.

Note: Be careful when specifying, in the node command, special characters (for example, `*` or `?`) which may be interpreted and substituted by Perl. You should quote the command string as a whole.

- **NMSCmdRecvFullReply([-iname=><name>,] [-out])**
  Waits for and receives the complete reply to the previous node command. The replied text is available in the `NMSEPI_OUTPUT_TEXT` module variable (and returned as a string if `-out` is used). The received text is also available to the `NMSCmdPatternMatch` command.

  Example

```
NMSCmdSendCommand("myOA", "dir");
if ( NMSCmdRecvFullReply() ) {
  system("echo \"$NMSEPI_OUTPUT_TEXT\" \
           | grep \"pe\"");
...
```

- **NMSCmdRecvNextLine([-iname=><name>,] [-out,] [<timeout>])**
  Waits for and receives the next line of the last reply of the issued node command. A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts a no-wait poll. The replied text is available in the `NMSEPI_OUTPUT_TEXT` module variable (and returned as a string if `-out` is used). The received text is also available to the `NMSCmdPatternMatch`, `NMSCmdGetNumColumns`, and `NMSCmdGetColumn` commands. The previous example can be rewritten as follows:

```
NMSCmdSendCommand("myOA", "r78 d");
while ( NMSCmdRecvNextLine() ) {
   if ( "$NMSEPI_OUTPUT_TEXT" =~ ".*pe.*" ) {
      print "$NMSEPI_OUTPUT_TEXT\n"
...
```

- **NMSCmdRecvNextChunk([-iname=><name>,] [-out,] [<timeout>])**
  Waits for and receives the next chunk of the last reply of the issued node command. A chunk is most efficiently processed by EPI and may contain

multiple lines of text (it can even end in the middle of a line). A timeout can be specified (by default, it waits forever). With a timeout of `0`, the command acts a no-wait poll. The replied text is available in the `NMSEPI_OUTPUT_TEXT` module variable (and returned as a string if `-out` is used). The received text is also available to the `NMSCmdPatternMatch`, `NMSCmdGetNumColumns`, and `NMSCmdGetColumn` commands. The previous example can be rewritten as follows:

```
NMSCmdSendCommand("myOA", "r78 d");
while ( NMSCmdRecvNextChunk() ) {
   if ( "$NMSEPI_OUTPUT_TEXT" =~ ".*pe.*" ) {
      print "$NMSEPI_OUTPUT_TEXT\n"
...
```

- **NMSCmdSkipRestOfReply([-iname=><name>])**
  This combination call waits for and ignores all replies until the end of response.

- **NMSCmdGetNumColumns([-iname=><name>])**
  Returns the number of blank (space or tab) separated columns in the previously received reply line. The column count is returned by the routine instead of an error code as well as in the `NMSEPI_NUM_COLUMNS` module variable.

- **NMSCmdGetColumn([-iname=><name>,] [-out,] <column> [, "..."] [, <target>])**
  Without `<target>`, returns the indicated blank (space or tab) separated column of the previously received reply line as the `NMSEPI_OUTPUT_COLUMN` module variable (and returned as a string if `-out` is used). If '`...`' is specified, all of the remaining columns starting at the given column are returned (blank separated). Column indexes start at `1`.

  Example

```
NMSCmdSendCommand("myPassp", "pp1 d fruni/101");
while ( NMSCmdRecvNextLine() ) {
   NMSCmdGetColumn(1);
   $att = "$NMSEPI_OUTPUT_COLUMN";
   NMSCmdGetColumn(3, "...");
   $val = "$NMSEPI_OUTPUT_COLUMN";
```

```
   if ( “$att” eq “operationalState” ) {
      print “State: $val\n”;
...
```

If <target> is specified, this command also checks whether the column has the same contents as the target and returns accordingly.

Example

```
NMSCmdSendCommand(“myPassp”, “pp1 d fruni/101”);
while ( NMSCmdRecvNextLine() ) {
   if ( NMSCmdGetColumn(1, “operationalState”) ) {
      print “State: “ . NMSCmdGetColumn(-out, 3,
               “...”) . “\n”;
...
```

- **NMSCmdPatternMatch([-iname=><name>,] [-out,] [“-g”,] <pattern> [, <substitute>])**
  This is the functionality of NMSEPIPatternMatch applied to the last received command response (full, line or chunk).

  Example

```
NMSCmdSendCommand(“myOA”, “dir”);
while ( NMSCmdRecvNextLine() ) {
   if ( NMSCmdPatternMatch(“.*pe.*”) ) {
    print “$NMSEPI_OUTPUT_TEXT\n”;
...
```

- **NMSCmdSendDestRequest([-iname=><name>,] [OA|GROUP|ALL])**
  This special utility command requests a list of all OA or Passport Group (or both) types of available routes. It corresponds to the cmccmd list command.

- **NMSCmdRecvNextDest([-iname=><name>,] [-out,] [<timeout>])**
  Waits for and receives the next reply to an NMSCmdSendDestRequest command. The reply is returned with the following module variables:

  NMSEPI_OUTPUT_TEXT
  is the full reply text line.

  NMSEPI_DEST_NAME

is the route name.

NMSEPI_DEST_TYPE
is the route type (OA or GROUP).

NMSEPI_DEST_STATE
is the route connection state (CONN, AUTH, or -).

Example

```
NMSCmdSendDestRequest("GROUP");
while ( NMSCmdRecvNextDest() ) {
   NMSCmdGetColumn(2);
   if ( "$NMSEPI_OUTPUT_COLUMN" eq "CONN" ) {
      ... # route is available
```

The following command is supported only in Perl EPI. It does not function when the graphical Tk extension is used (explicit Recv routines must be used instead).

- **NMSCmdBindCallback([-iname=><name>,] [-chunk|-ppcomp,] <callback command>)**
  Binds the specified Perl command string (typically a function invocation and its arguments) to the command interface. Execute this command string whenever a new line (default), a new chunk (-chunk), or a new Passport component (-ppcomp) is specified or if a reply is received from the server for this interface. The following environment variables are available to the callback:

  NMSEPI_INAME
  is the callback command interface name.

  NMSEPI_CB_REASON
  is the reason for the callback (LOST_CONNECTION, ERROR, ENDRESP, or RESPONSE or FLOW CALLBACK).

  NMSEPI_OUTPUT_TEXT
  is the next line (default) or chunk (-chunk mode) of output from the command.

  NMSEPI_PPCOMPID

is the the component ID of the next Passport component (`-ppcomp` mode).

The text manipulation commands previously described (`NMSCmdGetNumColumns`, `NMSCmdGetColumn`, and `NMSCmdPatternMatch`) are also available in the context of the callback in line and chunk mode. In Passport component mode, use `NMSCmdGetPPCompID, NMSCmdResetPPCompAttrs, NMSCmdGetFirstPPCompAttr, NMSCmdGetNextPPCompAttr,` and `NMSCMDFindNextPPCompAttr` to extract the Passport component information. The event loop is launched with `NMSEPIEventLoop` and never returns.

Example

```
sub mycb {
   # ...Process the reply...
}
...
NMSCmdSendCommand("myOA", "r72 q serv");
NMSCmdBindCallback("mycb");
...
NMSEPIEventLoop();  # never returns
```

- **NMSCmdRecvNextPPComp([-iname=><name>,] [-out,] [-var=><var name>,] [<timeout>])**

  `NMSCmdRecvNextPPComp` is a form of `NMSCmdRecvFullReply` that waits for and receives the next Passport component reply to the previous command (i.e. the result of a `list` or `display` command). The name of the received component is available in the `NMSEPI_PPCOMPID` variable (and returned as a string if `-out` is used) and any error message is available in the `NMSEPI_OUTPUT_TEXT` variable.

  Note: Make sure you always specify the `-notab` (no tabular output) CAS command line option when sending a Passport CAS display command with wildcards if this command is to be used to extract the replies.

  The `NMSCmdGetPPCompID`, `NMSCmdGetNextPPCompAttr`, and `NMSCmdFindNextPPCompAttr` commands can then be used to extract the replied component information. In addition, if a variable is specified

with the `-var` option, the named variable is set as a hash array whose elements are the individual component attribute values. The element key is the returned attribute name. If the attibute is a list, vector or array, the first index is added to the key with a coma as separator (for example, `$rep{pktFromIfByPrio,ep0}`). For two dimensional arrays, the entry is created with the attribute name as key and the column title as value. Another entry is created with "`<attribute name>,<row title>`" as key and the list of column labels as value. The remaining entries for this attribute have "`<attribute name>,<row label>`" as index and the list of corresponding columns as values. Finally, the entry key can also be one of the following special values; `Message`, contains any message emitted by Passport not part of of an attribute value (i.e. error messages), `CompID`, is the returned component's name.

It is possible to list all the array keys using Tcl's "`array names <array name>`" construct.

Example

```
NMSCmdSendCommand("myGroup", \
   "TOTO display shelf card/* utilization");
while ( NMSCmdRecvNextPPComp(-var=>reply) ) {
   print "Card: $NMSEPI_PPCOMPID\n";
   print "Average CPU: $reply{cpuUtilAvg}\n";
...
```

Other examples, using a display of the `Module-Vcs` and Passport `Shelf-Card` components, are:

```
# Simple attribute:
   $reply{cpuUtil} ... "5 %"
# SET value
   $reply{highPriorityPacketSizes} ..
               "16 32 64 128 256..."
# Vector values
   $reply{memoryUsage,fastRam) = "0 kbyte"
   #... and so on ...
# 2-D array values
   $reply{windowSize} ... "throughputClass"
   $reply{windowSize,packetSize} ...
               "0 1 2 3 4 ..."
   $reply{windowSize,16} ... "4 4 4 4 4 ..."
   #... and so on ...
```

```
# error
   $reply{Message} ... "   Component is disabled."
   $reply{localMsgBlockCapacity} ... "? kbyte"
```

- **NMSCmdGetPPCompID([-iname=><name>,] [-out,] [-var=><var name>])**

  `NMSCmdGetPPCompID` extracts the name of the last received Passport component in the `NMSEPI_PPCOMPID` variable. If `-out` is specified, the name is also returned as a string. If an associative array variable name is specified with `-var`, the array is also filled with the component's attributes as described for the `NMSCmdRecvNextPPComp` command. Finally, the list of Passport component attribute is reset to the beginning for the `NMSCmdGetNextPPCompAttr` and `NMSCmdFindNextPPCompAttr` commands.

  Example

```
...
while ( NMSCmdRecvNextPPComp(-iname=>alt, \
              -var=>reply) ) {
   NMSCmdGetPPCompID(-iname=>alt);
   print "Component: $NMSEPI_PPCOMPID";
...
```

- **NMSCmdGetNextPPCompAttr ([-iname=><name>,] [-out])**

  `NMSCmdGetNextPPCompAttr` extracts the next attribute from the last received Passport component. The attribute name (and index, see the discussion on the hash array in `NMSCmdRecvNextPPComp`) is returned as the `NMSEPI_PPATTR_NAME` and its value as the `NMSEPI_PPATTR_VALUE` variables. If `-out` is specified, both values are also returned as a single string with a space as separator.

  Example

```
...
while ( NMSCmdGetNextPPCompAttr() ) {
   print "Attribute: $NMSEPI_PPATTR_NAME";
   print "Value: $NMSEPI_PPATTR_VALUE";
...
```

- **NMSCmdFindNextPPCompAttr(-iname=><name>,] ["-out",] <attribute name>)**

  `NMSCmdFindNextPPCompAttr` extracts the next attribute from the last received Passport component whose name (in index, see the discussion

on the associative array in NMSCmdRecvNextPPComp) matches the specified value. The attribute name/index is returned as the NMSEPI_PPATTR_NAME and its value as the NMSEPI_PPATTR_VALUE variables. If -out is specified, both values are also returned as a single string with a space as separator.

Example

```
...
if ( NMSCmdFindNextPPCompAttr("bytesSent") ) {
    NMSEPILongArith($prev, "-", $NMSEPI_PPATTR_VALUE);
    print "Delta: $NMSEPI_RESULT";
    prev = $NMSEPI_PPATTR_VALUE;
...
```

- **NMSCmdDoCommandFile [-iname <name>] [-out,]**
  [-trace[:[C][R][E][X]],] [-test] [-cb,]
  [-bestEffort,] [-avl=><var name>,]
  [<var>=<value>, ...] <dest>, <file path>)
  NMSCmdDoCommandFlow([-iname=><name>,] [-out,]
  [-trace[:[C][R][E][X]],] [-test] [-cb,]
  [-bestEffort,] [-avl=><var name>,]
  [<var>=<value>, ...]
  <dest>, <command flow>)
  NMSCmdTerminateCommandFlow([-iname=><name>,]
  [<result>])

  These calls support the synchronous execution (commands and replies) of command flows from the file identified by <filePath> or the string identified by <commandFlow>. A command flow is a sequence of device commands (one per line) to be executed as one. The file may be specified as a formal path (starting with '/', '.' or '. .', or '~' which is automatically substituted to the user's HOME account), or as a relative path in which case the file is automatically searched for; first, in $HOME/MagellanNMS/<file path>, then /opt/MagellanNMS/cfg/<file path>, and finally /opt/MagellanNMS/lib/<file path>.

  All commands are executed through the named destination (if empty -- ""-, the destination must be prefixed to every device command in the flow). The execution of the flow terminates at the first failure unless -bestEffort is specified. Variable substitution on each command in

the flow is supported with the hash array named with `-avl` as the source of the variable name-value pairs. Additional (or overriding) variable assignments can be specified with `<var>=<value>` arguments.

The flow may produce some output text and numerical results available as the `NMSEPI_OUTPUT_TEXT` and `NMSEPI_RESULT` global variables respectively. The output text is produced from the flow using the `@PRINT` directive (see below). If `-trace` is specified the executed device commands are also added to the output text. This output contains a `#*` prefix for plain commands and `#?` for conditional commands followed by their output followed by their output and a `#? END`, `#? FAILED`, `#* END`, or `#* FAILED` to indicate the end of of the corresponding command. The output result is produced from the flow by the `@EXIT` directive or the `NMSCmdTerminateCommandFlow` command in callback mode (see below). You can control specific tracing by using the following optional modifiers:

C traces only the executed plain commands. Control construct commands (`@SWITCH`, `@FOREACHPP`, ...) are not traced.

R traces only the plain command responses (also excludes control constructs).

E traces only errored responses from plain commands (also excludes control construct commands).

X traces control construct commands and responses

You can specify one or more such modifier letters following a colon. You can also control tracing from the flow itself by using the `@TRACE` construct.

The output processing command (`NMSCmdGetNumColumns`, `NMSCmdGetColumn`, and `NMSCmdPatternMatch`) can also be used to examine the output.

If `-cb` is specified, the output text is not returned in the `NMSEPI_OUTPUT_TEXT` variable. Instead, the callback bound to the command interface by `NMSCmdBindCallback` is invoked for each line of output (including the `@PRINT` and `#?` and `#*` comments) where it is

available in the usual manner (with reason `RESPONSE`). From the callback, it is possible to force the termination of the flow execution by invoking the `NMSCmdTerminateCommandFlow` command with the desired numerical result (as if `@EXIT` had been reached in the file).

Note that in callback mode, even though the output is returned line by line through the bound callback, the execution of the flow is still synchronous and no other EPI actions will be performed until the flow execution has completed. In other words, the flow/file execution function will not return until the flow execution is complete during which the bound callback will be invoked for each output line. Once the flow execution completes, the bound callback is called with reason `ENDRESP` on success or ERROR on failures, unless the execution was explicitely terminated with `NMSCmdTerminateCommandFlow`.

If `-test` is specified, the command flow/file is executed in test mode as described later

Command flows consist of a list of device commands with, optionally, substitution variables identified by a dollar sign '`$`' (to specify a plain `$`, escape it as '`\$`'). For example, the following Passport command sets the `committedInformationRate` of a FrameRelay DLCI (read as one line):

```
$name set FrUni/$fruni Dlci/$dlci Sp\
            committedInformationRate $cir
```

A flow containing such a line should be executed with an attribute-value

associative array containing at least:

```
%avl = (    name=>NODER16, fruni=>120,
            dlci=>25, cir=>56000 );
```

Other forms of substitution variable specification include;

${<variable name>**}**	same as without the brackets.
$!**<variable name>, or **${!<variable name>**}**	(strict substitution) When used in device commands (does not apply to the special directives below), the command will be skipped (silently not executed) if the specified variable has no associated value or is empty. Note: This form is only available in actual device commands.
${<variable name>**:-** <default value>**}**	If the specified variable has no associated value or is empty, substitutes the specified default value instead.
$%	This special internal variable holds the contents of the last executed `@SWITCH` command or the value of the matched `\(\)` subpattern of the last executed `@case` command (see below)
$%%	This special internal variable holds the pattern-matched contents (whole or sub-pattern) of the last executed `@IF` command (see below).

$?	This variable contains the numerical result of the last issued macro (@DO) or flow (@INCLUDE/@RUN).
${<variable name>**[**<index>**]}**	This represents an associative array value in the Flow language itself (not to be mistaken by array values in the scripting language). Such values can be directly set (@SET, @DEFINE, @LOCAL) or provided by specialized commands (@FOREACHPP, @SPLITCOMP, @SPLIT). When used, both the variable name and the index can be also be substituted variables (for example, ${array[$i]} in a loop that increments the value of $i). The ! and :- constructs also apply to the array entry (for example, ${array[$i]:-0} defining 0 as the default value for the entry).

Command flows also support special processing directives as indicated int the following table:

@PRINT <string>	appends the specified text (after variable substitution) to the command output (NMSEPI_OUTPUT_TEXT).
@FORMAT <multi-line string...> **@END**	same as @PRINT but for a multi-line piece of text.
@EXIT [<code>]	terminates the flow's execution with the specified result code (NMSEPI_RESULT). If no @EXIT directive is executed, the flow will have its result set to 0 on success or 4 if the flow was terminated by a failed command.

@RETURN [<code>]	like @EXIT but when invoked from an included file (see @INCLUDE), only terminates the execution of the included file and returns to the calling flow/file.
@TRY [<command>]	executes the command and ignores its possible failure, even if -bestEffort was not specified. If the command is omitted, sets the current operating mode for subsequent commands as if -bestEffort had been specified. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.
@CRITICAL [<command>] or **@CRIT** [<command>]	executes the command and terminates the flow if the command fails, even if -bestEffort was specified . If the command is omitted, sets the current operating mode to critical for subsequent commands. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.
@TRACE [<command>]	traces this command, even if -trace is not specified. If the command is omitted, sets the current operating mode to trace (as if -trace had been specified) for subsequent commands. Other modifiers (@TRY/@CRITICAL, @LOG/@NOLOG) may also be specified.
@NOTRACE [<command>]	does not trace this command, even if -trace is specified. If the command is omitted, sets the current operating mode to no-trace for the subsequent commands. Other modifiers (@TRY/@CRITICAL, @LOG/@NOLOG) may also be specified.

@LOG [<command>]	logs this command as long as a log file has been defined (NMSCmdOpenCommandLog or @LOGFILE). If the command is omitted, sets the current operating mode to logging. Other modifiers (@TRY/@CRITICAL, @TRACE/ @NOTRACE) may also be specified.
@NOLOG [<command>]	does not log this command, even if logging was enabled. If the command is omitted, sets the current operating mode to no-logging for the subsequent commands. Other modifiers (@TRY/ @CRITICAL, @TRACE/@NOTRACE) may also be specified.
@LOGFILE [[**+**]<log file path>]]	This controls command logging (see NMSCmdOpenCommandLog). Without arguments, this is equivalent to @LOG. If the log file is identified, the issued commands are logged to it from this point on (unless modified by @NOLOG). If the file path is prefixed with the plus (+) sign, the logs are appended to the file if it exists (else the file is overwritten with the new logs).

```
@IF <var.> [==|!=
            <patterns>]
or
@IF <var.> <|>|<=|>=
            <value
<commands>
...
[@ELSE
<commands>
...]
@END
```

evaluates the specified variable expressions and executes the first command block if it finds that it matches one (==) or does not match any (!=) of the patterns, first form, or compares (numeric or string, as appropriate), the second form, to the specified value. Otherwise, the command block following the @ELSE directive is executed, if any. If just the variable expression is specified, the test is positive if the expanded value is not empty. If the test was a pattern matching one, the value of the sub-pattern is available as the $%% internal variable.

```
@FOR <var> <from> <to>
       [<increment>]
or
@FOR <var> IN
       <token list>
<commands>
...
@END
or
@FOR <var> KEYS
       <array name>
<commands>
...
@END
```

executes the command block repeatedly, in the first form, while incrementing the named (<var>) numerical variable in the AVL from <from> to <to> in jumps of <increment> (defaults to 1), or in the second form, iterating the varaible across the list of blank separated tokens. The variable ($<var>) can be used in the command block. The loop can be terminated prematurely by calling the @BREAK command from within the command block. Loops can be nested. The third form allows the variable to scan the existing indicies of the named associative array (for example, the Passport attribute values from @FOREACHPP).

@FOREACHPP <var> <Passport list command> <commands> ... **@END**	executes the specified Passport list command, and then iterates over the returned component names, assigning its name to the named variable and executing the command block. You can terminate the loop by calling the @BREAK command from within the command block. Loops can be nested. If the Passport command was a display one, the extracted attribute values are also available as the entries of an associative array of the same name as the specified variable. They are provided in the same way as from the NMSCmdRecvNextPPComp function. If wild-cards are used, make sure you specify the -notab display command option.
@FOREACHLN <var> <command> <commands> ... **@END**	executes the specified command command then iterates over the returned output line by line, assigning each one to the named variable and executing the command block. You can terminate the loop by calling the @BREAK command from within the command block. Loops can be nested.
@BREAK	invoked from within a @FOR/ @FOREACHPP/@FOREACHLN/ @WHILE/@WHILDO loop construct, it terminates the enclosing loop prematurely. In other situations, it acts like @RETURN with no return code.

@CONTINUE	invoked from within a @FOR/ @FOREACHPP/@FOREACHLN/ @WHILE/@WHILDO loop construct. This command causes the loop to immediately iterate to the next cycle. Like @BREAK, the following loop-code is not executed but unlike @BREAK, the loop is not abandoned.
@CB <text>	if the Flow is running in callback mode (-cb option), invoke the callback with the specified text as the output text (NMSEPI_OUTPUT_TEXT). The callback reason (NMSEPI_CB_REASON) is then set to FLOW_CALLBACK. The callback may interpret this text as convened and, in reply, set or reset AVL variables with NMSCmdSetFlowCBAVL before returning so the flow can use the results. You can use callback to query another system or a user for a value needed in the flow processing (Wizzard).

```
@SWITCH <test command>
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

executes the test command and executes the commands following the first `@CASE` whose patterns match the output of the test command (the optional commands that follow the `@SWITCH` before the first `@CASE` are always executed). Only one `@CASE` block in the `@SWITCH` construct is executed. `@SWITCH` blocks can be nested. Patterns are specified in GREP format with a '`|`' separating alternatives. If no pattern is specified, the `@CASE` block accepts any output. The `@SWITCH` command is traced to ouput, if enabled, as:

```
#? <test command>
<command output...>
#? END
```

(If the command could not be executed, the last line will be **`#? FAILED`** instead).

The `@CASE` patterns may contain one subexpression (`\(\)`), each of whose matched value is available in the following code as the `$%` substitution variable. The usual modifiers (`@TRY`/`@CRITICAL`, `@TRACE`/`@NOTRACE`, `@LOG`/`@NOLOG`) can be specified after the `@SWITCH`.

```
@SWITCHVAL <value expr.>
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

this construct behaves much like `@SWITCH` but instead of getting its pattern match target value from a device command output, that value is directly specified as a parameter.

@WHILE <var.> [==|!=|<|>|<=|>= <patterns/value>] <commands> ... **@END**	repeatedly executes the command block as long as the text (same as @IF) succeeds. The loop can be broken prematurely by invoking @BREAK.
@INCLUDE <file path> or **@RUN** <file path>	the named file is included and processed as though its contents were part of the current flow. The difference betwen @INCLUDE and @RUN is that variable definitions (@DEFINE) performed in the file invoked by @INCLUDE apply to the calling flow. Variable definitions in a file invoked by @RUN are forgotten upon return. If the file is identified as a relative path, the standard Preside Multiservice Data Manager (MDM) search path applies (see above). If the file cannot be read, the flow's execution terminates unless -bestEffort was specified or the @TRY prefix is used. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified before @INCLUDE/@RUN.

@MACRO <name> [<parameters...>] <code text> ... **@ENDMACRO**	defines a new macro thatmacro that can be executed later with @DO/@IFDO/@WHILEDO. The macro can be given a number of positional argument names to be provided when executed (the last specified parameter name gets all the remaining arguments passed). These parameter names have local scope to the macro (as if defined with @LOCAL). The macro can be recursive. Just like @INCLUDE/@RUN, it can be terminated by @RETURN which specifies a return code available as the $? variable to the caller. Variables defined/set by the macro have the same scope as the caller unless defined with @LOCAL. Macros must be defined before they are used. The macros themselves have global scope and can be redefined.
@DO <macro name> [<arguments...>]	invokes the named defined macro. The specified arguments are assigned (local scope to the macro) to the macro's parameter names -- the name gets the left over arguments). The @RETURNed value from the macro is available as the $? internal variable. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified before @DO.

```
@IFDO <macro name>
      [<arguments...>]
<command>
...
[@ELSE
<commands>
...]
@END
```

merges the functionality of the @IF and @DO constructs. Executes the macro as for @DO then performs either the first command block if the macro returns a `0` result, else performs the second (@ELSE) command block if any. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @IFDO.

```
@WHILEDO <macro name>
      [<arguments...>]
<commands>
...
@END
```

merges the functionality of the @WHILE and @DO constructs. Repeatedly executes the macro as for @DO then the command block as long as the macro returns a `0` result. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @WHILEDO. As with @WHILE, the loop can be broken with @BREAK invoked in the command block.

```
@DEFINE <name> <value>
```

defines or redefines a variable name. The new value is available in the current command block and the blocks it invokes, notably for included files. For example, if a @DEFINE is invoked in a @CASE block, the modified value applies to the commands in that block but not in the commands that follow the @SWITCH construct for that @CASE. Similarly, @DEFINEs used in included files have no effect on the calling command block.

@UNDEFINE <name>	Contrary to @DEFINE, @SET, and @LOCAL, undefines the named variable. All matching associative array values are also undefined. To undefine a single entry in the array, specify its full name (for example, @UNDEFINE <array>[<index>]).
@LOCAL <name> [<value>]	Like @DEFINE but the new variable has local scope (it will not replace nor exist in the caller's scope, for example when used from within a macro or included flow). The local scope also applies to associative arrays by that name. This is useful when defining macros that need varaibles for which you do not want to override existing values.
@SET <name> <value1> [**+|-|*|/|%|~**<value2>]	like @DEFINE but sets the variable to the result of the numerical expression (+, -, *, /, % -- remainder). The ~ operator performs a pattern match using the second value as a GREP style pattern pattern list (| separated). The variable is set to the matching portion or to nothing. If the pattern contains a \(\) delineated sub-pattern, it is that matching sub-pattern that is used as the new value. If only the name and first value are specified, the effect is the same as @DEFINE.

@SPLIT[**(**<separators>**)**]
<array name>
<string>

This tokenizes the specified string. The individual tokens are assigned to indexed elements of the named associative array (starting at 1). The actual variable's value is the number of resulting tokens. By default, tokenization is done on blanks. Alternatively, the separator characters can be provided between brackets. For example, the following will print the individual applications in a Passport AVL, one per line:

```
@SPLIT(, \t\n) app $avl
@FOR i 1 $app
     @PRINT ${app[$i]}
@END
```

@SPLITCOMP <array name> <component ID>

analyzes the specified component ID and provides the results in the named associative array. The following examples are based on the component EM/TOTO LP/2 Ds1/0.

$<array>
: the component ID in API format (EM TOTO LP 2 DS1 0)

${<array>**[_MOD]}**
: the module name (TOTO)

${<array>**[_SUB]}**
: the subcomponent portion (minus first level) (LP/2 DS1/0)

${<array>**[_PAR]}**
: the parent subcomp portion (minus last level) (EM/TOTO LP/2)

${<array>**[**<category>**]}**
: the relative instance value for that level (EM -> TOTO, LP -> 2, DS1 -> 0)

@WAIT <nb seconds>

blocking wait for the specified number of seconds.

```
@ASK <name>
     [:E|:I|:S]
     [/validation
patterns/]
     [=<default value>]
     <prompt string>
@CASK <name>
     [:E|:I|:S]
     [/validation
patterns/]
     [=<default value>]
     <prompt string>
```

asks you (standard input) for the value of the named variable using the specified string as a prompt. The expected type of the value can be specified as one of the following for a plain string:

`:E` for a string token enumeration
`:I` for an integer
`:S` (the default)

You can use a pattern between two forward slashes (//), vertical bars (|). Specify the pattern so that EPI validates the entered value and prompts if there is no match. For enumeration, the pattern is a blank/coma separated list of words to match (for example, `/on off/`). For integers, the pattern is a blank/coma separated list of numeric values or ranges (for example, `/1,3,5,10-15,20/`). For strings (default) the pattern is an extended GREP style pattern list with `|` between alternative patterns (for example, `/.* Ds1\/.*|.* E1\/.*/` -- The forward `/` in the pattern is escaped with a single backwards \ so it is not included as the end of the pattern). If you specify a default value, this value is set to the variable if you enter nothing (carriage return only). If you do not specify a default value, you are prompted when you enter an empty string.

`@CASK` (conditional ask) is similar to `@ASK` except that it does not prompt if the variable already has a non-empty value..

`# <comment>`	comment line.

`<device command>`	actual device command, optionally with embedded variables (`$` prefix) invoked a‘fter substitution. If the command fails, the flow execution terminates (no `-bestEffort` nor `@TRY` prefix).
	The command is traced to ouput, if enabled, as:
	`#* <device command>`
	`<command output...>`
	`#* END`
	(If the command indicated an error, the last line will be `#* FAILED` instead).

Note: Note that `@TRY`, `@CRITICAL`, `@TRACE`, and `@NOTRACE` can be combined. They can also be used with the `@INCLUDE` directive. Also, the command specified with `@SWITCH` can also start with `@TRACE` or `@NOTRACE`.

Example
Assuming a file (examp.tmpl) containing:

```
# Frame Relay QOS example
@SWITCH $name l FrUni/$fruni Dlci/$dlci
@CASE failed|ERROR
  @PRINT $name FrUni/$fruni Dlci/$dlci does not exist!
  @EXIT 1
@CASE
  $name set FrUni/$fruni Dlci/$dlci Sp cir $cir
  $name set FrUni/$fruni Dlci/$dlci Sp bc $bc
  $name set FrUni/$fruni Dlci/$dlci Sp be $be
  @IF $lmi
    $name set FrUni/$fruni Lmi procedures $lmi
    $name set FrUni/$fruni Lmi side $lmiside
  @END
@END
```

This flow can be executed with:

```
if ( NMSCmdDoCommandFile(-avl=>avl, \
                "*", examp.tmpl) ) {
    print "Failed!!!\n$NMSEPI_OUTPUT_TEXT\n";
    exit 1;
}
```

If, instead, the contents of the file above are stored/built in a shell variable (for example, FLOW_STRING), the flow can then be executed with the following (note the quotes around the $FLOW_STRING specification to avoid the shell from absorbing the carriage returns between commands):

```
if ( NMSCmdDoCommandFlow(-avl=>avl, \
                "*", "$FLOW_STRING") ) {
    print "Failed!!!\n$NMSEPI_OUTPUT_TEXT\n";
    exit 1;
}
```

Example

The following are small examples of flow code usage:

```
# determine the Passport version and save it
# for later use
@SWITCH $name d software avl
@CASE base_\([^ ,]*\)
  # $% contains the last \(\) match (the base version)
  @PRINT Passport version : $%
  @DEFINE ppversion $%
  # set variables accordingly (Tm subcomponent
  # introduced?)
  @IF $ppversion == CA.*|CB.*
    @DEFINE tm Tm
  @ELSE
    @DEFINE tm
  @END
@END
...
# use the variable defined above and set the
# p1 parameter to a default value if not set
$name set AtmIf/$atm Vcc/$vpci Vcd $tm txTdp 1 \
                ${p1:-64000}
...
```

```
# create multiple DS1 channels with @FOR
@FOR chan 0 24
  $name add Lp/$lp DS1/$ds1 Chan/$chan
  $name Lp/$lp DS1/$ds1 Chan/$chan timeslots $chan
  # run a secondary flow to create a FrUni
  # for each channel (the flow has access to
  # the current AVL including the $chan variable)
  @RUN addFrUni
@END
```

CAUTION

DoEPITemplate

The DoEPITemplate helps you use and invoke command flows by handling scripting aspects. You only need to create the required flow text. The utility handles everything else, including the Passport configuration pre and post-amble for the configuration flows.

See "DoEPITemplate Utility" (page 607) for the description of the DoEPITemplate utility.

Test Mode

To test command files/flows without executing them, for example, while developing complex configuration templates, you can invoke the command with the `-test` option. This allows you to test the variable substitution, the `@SWITCH`/`@CASE` pattern matching and the general execution flow of the commands with or without actually executing them. In test mode, commands to be excuted are traced to the standard error stream (after variable substitution and prefixed by its line number) then a prompt usually asks for confirmation if it should be executed. The prompt depends on the command being executed:

@FOR <variable name> <from> <to> [<increment>]
or
@FOR <variable name> **IN** <token list>
on each iteration, the prompt offers to execute the command block or not::

```
> Confirm command block execution? ([y]|n|q)>
```

If `q` is answered, the flow's execution is terminated as though an `@EXIT` directive had been encountered. If `y` is answered (the default), the

command block is executed as normal. If n is answered, the loop's execution is terminated as though a @BREAK directive had been encountered.

@SWITCH <test command>
the prompt offers to execute the command or not:

> Execute it? (y|[n]|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered, the command is executed as normal. If n is answered (the default), you are then prompted for the output the command would have produced in order to test the @CASE pattern matching:

> Command reply? (end with @@)>

The output should be terminated with a line containing only the @@ characters. If you do not want to test the pattern matching, just @@ and you will be prompted for confirmation of the match for each executed @CASE directives.

@CASE [<patterns>]
the prompt indicates if the pattern matching would succeed:

> Matches, confirm command block execution? ([y]|n|q)>

or fail:

> Does not match, execute command block anyways? (y|[n]|q)>

offering to execute the following command block or not.
If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default on success), the command block is executed as normal. If n is answered (the default on failure) the flow is executed as though the @CASE pattern would not match and the next @CASE is block, if any, is tried instead.

@IF <variable> [==|!= <patterns>]
the prompt indicates if the test succeeds:

> Test succeeded, confirm command block execution? ([y]|n|q)>

or fails:

> Test failed, execute command block anyways? (y|[n]|q)>

offering to execute the following command block or not.
If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default on success), the command block is executed as normal. If n is answered (the default on failure) the flow is executed as though the @IF test failed and the @ELSE is block, if any, is tried instead. If this block is executed, the @ELSE block, if any, will be ignored.

@ELSE
the prompt offers to execute the following block:

```
> Confirm command block execution? ([y]|n|q)>
```

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command block is executed as normal. If n is answered, the command block is skipped until the corresponding @END construct.

@INCLUDE <command file>, or
@RUN <command file>
the prompt offers to confirm the execution of the file:

```
> Include/Run the file? ([y]|n|q)>
```

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command file is executed as normal. If n is answered, the command file is not included.

<device command>
the prompt offers to execute the command or not:

```
> Execute it? (y|[n]|q)>
```

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered, the command is executed as normal. If n is answered (the default) the command is not executed and the next command is tried. The behaviour is the same for commands prefixed with @TRACE, @NOTRACE, @TRY, and @CRITICAL.

@PRINT <string>
@EXIT [<code>]
@DEFINE <name> <value>
@END
The command is echoed as is without further prompts.

- **NMSCmdSetFlowCBAVL [-iname <name>]**
  <var name> <value> ...
  called within a callback (`NMSCmdBindCallback`). Invoke this command from a command flow executing (`NMSCmdDoCommandFile` or `NMSCmdDoCommandFlow`) by the `@CB` construct. The callback can then inform the flow of the results by setting or resetting some variable assignments. You can specify multiple variable name/value pairs.

- **NMSCmdOpenCommandLog [-iname <name>]**
  [-append]
  <log file path>
  Opens the specified file and logs all commands executed with `NMSCmdSendCommand`, including those issued by a command flow or file. Each device command executed by the flow is logged, not only the flow's execution. If you specify `-append`, the commands are appended to the file if it exists. By default the file is overwritten. The format of the log allows its replay as a command flow or as input to the `cmccmd` utility. The commands are logged from the module on, no group name, and with all variables substituted for flow commands). Note that logging can also be controlled from the flow itself through the `@LOG` and `@LOGFILE` constructs.

- **NMSCmdCloseCommandLog [-iname <name>]**
  Stops command logging and closes the log file opened by `NMSCmdOpenCommandLog`.

Customer Database access

The Customer Database access capabilities or EPI allow you to query and modify the contents of the Customer Databases managed by Preside Multiservice Data Manager (MDM). The Customer Databases are controlled by servers (`cdbserver`) and can be accessed through the Customer Database tool (GUI). Customer Databases can also be accessed by the `cdbextract` and `cdbmerge` utilities, which are more suited for bulk extraction and population. EPI provides more programmability of the database's contents. For example, by using EPI you can receive an alarm feed from GMDR and extract the matching customer information from the database. You can then use this information to generate a trouble ticket to an external system.

Customer Database Access commands are used in the following sequence:

1 Initialize a Cdb interface.

2 Connect to the Cdb server for the appropriate database.

3 Send Fetch, Store, or Erase commands.

4 Send Query commands and receive the matching replies either synchronously with the RecvReply command or asynchronously by binding a script callback to the Cdb interface.

5 Use Perl or the provided commands to analyze the Fetch and Query replies.

6 Disconnect from the Cdb server.

7 Drop the Cdb interface.

The Customer Database server is synchronous in that a Cdb interface can only perform one Fetch, Query, Store and Erase command at once. The Fetch, Store, and Erase commands are synchronous in that they always wait for the reply. The Fetch command also provides the matching information on return (similar to a non-wild card Query followed by a RecvReply command).

Customer Database Access commands

The following Customer Database Access commands are provided:

- **NMSCdbInit([-iname=><name>])**
  Initializes a Customer Database interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultCdb` to make it the default interface).

- **NMSCdbDrop(-iname=><name>)**
  Drops the named Customer Database interface (frees allocated resources).

- **NMSCdbConnect([-iname=><name>,] <CDB name>**
  [, <CDB server host>])
  Connects the interface to the CDB server for the indicated database and host (defaults to “localhost”)..

  Example

```
NMSCdbInit();
if ( NMSCdbConnect(“EastCustDB”, “bcarse88”) ) {
   ... # we’re connected
```

- **NMSCdbDisconnect [-iname=><name>]**
  Disconnects the interface from the CDB server. The interface may then be reconnected to another server.

- **NMSCdbFetch([-iname=><name>,] [-out,]**
  [-hier,]<component ID>)
  Queries the Customer Database for information on the specified component. Typically, the component name is specified in canonical format (see `NMSEPIConvertCompId`). The command blocks and waits for the reply. The reply is returned as global variables, and as a result if `-out` is specified. If `-hier` is specified and no match is found, the routine looks for matching data on the parent components. The `NMSEPI_CDB_COMPID` is also set to the component ID on which the match was found. With `-hier`, both the canonical and display format of the component IDs are searched for in the database. The following module variables are set:

  `NMSEPI_CDB_COMPID`
  is the component ID.

  `NMSEPI_CDB_RELCOMPID`
  is the associated component ID, if any, an empty string otherwise.

  `NMSEPI_CDB_DATA`
  is the associated textual data.

  `NMSEPI_CDB_DATE`
  is the associated date stamp (as in `YYYYMMDD`).

  `NMSEPI_CDB_SOURCE`
  is the associated three character source code.

  If `-out` is specified, the matching reply will be output with the following format (matches the output and input to the `cdbextract` and `cdbmerge` utilities):

  <source:3>;<date:8>;<component ID:65>;<related component ID:46>;<data length:4>;<data: variable>\n

Example

```
if ( NMSCdbFetch("EM TOTO FRUNI 45") ) {
   print "Data: $NMSEPI_CDB_DATA\n";
   ...
```

If the -out option had been specified, the command would have produced the following following string result (the text is split with '\' for readability):

```
EPI;19980508;EM TOTO FRUNI 45                \
                                  ;FRAD AP34    \
                                   ;0024;Client \
Number: AP0002345
```

- **NMSCdbQuery([-iname=><name>,]**
  -comp=><component ID pattern>
  | -rel=><component ID pattern>
  | -data=><data pattern>)

  Queries the Customer Database for information matching the specified grep-style pattern (see NMSEPIPatternMatch on page 250). Only one pattern may be specified to match the entry's component name (-comp), associated component name (-rel) or data (-data). The command immediately returns. The returned replies must be extracted synchronously with NMSCdbRecvReply or asynchronously through a script callback bound to the interface with NMSCdbBindCallback.

  Example

  ```
  if ( NMSCdbQuery(-comp=>"EM TOTO FRUNI .*") ) {
     ... # query sent successfully
  ```

- **NMSCdbRecvReply([-iname=><name>,] [-out,] [<timeout>])**
  Waits for and receives the next Query reply record from the CDB server. A timeout can be specified (by default, it waits forever). With a timeout of 0, the command acts as a no-wait poll. If -out is specified, the received record is returned as a string result. The return error code is in NMSEPI_RESULT (5 indicates a timeout occurred). The following module variables are set:

  NMSEPI_CDB_COMPID
  is the component ID.

NMSEPI_CDB_RELCOMPID
is the associated component ID, if any, an empty string otherwise.

NMSEPI_CDB_DATA
is the associated textual data.

NMSEPI_CDB_DATE
is the associated date stamp (as in YYYYMMDD).

NMSEPI_CDB_SOURCE
is the associated three character source code.

If -out is specified, the matching reply will be returned as a string result with the same format as for the NMSCdbFetch command.

NMSCdbRecvReply can be called for each reply and will return 0 (also in NMSEPI_RESULT) to indicate success. It will return 6 to indicate that no more replies are available. Further calls, as well as calls when there is no active Query command, will return an error indication.

Example

```
NMSCdbQuery(-data=>“Client Number: AP.*”);
while ( NMSCdbRecvReply() ) {
   print “Component: $NMSEPI_CDB_COMPID\n”;
   print “Updated:   $NMSEPI_CDB_DATE\n”;
   print “Data:      $NMSEPI_CDB_DATA\n”;
   ...
```

The following command is supported only in Perl EPI. It does not function when the graphical Tk extension is used (NMSCdbRecvReply must be used instead).

- **NMSCdbBindCallback([-iname=><name>,] <callback command>)**
  Binds the specified Perl command string (typically a function invocation with its arguments) to the Cdb interface. This command string will be executed whenever a Query command reply is received from the server for this interface. The following environment variables are available to the callback:

  NMSEPI_INAME
  is the API interface name of the callback.

NMSEPI_CB_REASON
is the reason for the callback (LOST_CONNECTION, ERROR, ENDRESP, or RESPONSE).

NMSEPI_CDB_COMPID
is the component ID.

NMSEPI_CDB_RELCOMPID
is the associated component ID, if any, an empty string otherwise.

NMSEPI_CDB_DATA
is the associated textual data.

NMSEPI_CDB_DATE
is the associated date stamp (as in YYYYMMDD).

NMSEPI_CDB_SOURCE
is the associated three character source code.

```
sub mycb {
   # ...Process the reply...
}
...
NMSCdbQuery(-data=>"AP.*");
NMSCdbBindCallback("mycb");
...
NMSEPIEventLoop(); # never returns
```

- **NMSCdbStore([-iname=><name>,] <component ID>, <data>**
  [, -rel=><related component ID>]
  [, -date=><date as YYYYMMDD>]
  [, -source=><source as SSS>])

  Stores the specified information in the Customer Database. Typically, the component name is specified in canonical format (see NMSEPIConvertCompId). The information is added, or replaces that already associated with the component. The component name and data must be specified. An associated component name (-rel), a date stamp

(`-date`, defaults to the current date), and a 3-character source code (`-source`, defaults to `EPI`) can also be specified. The command is synchronous since it blocks and waits for the reply from the CDB server.

Example

```
if ( NMSCdbStore(“EM TOTO FRUNI 43”, “Client Number:
DP542
Contact: (613) 763-2211)”, -rel=>“FRAD 213”, -
source=>“GCG”) ) {
   ... # data is now stored
```

- **NMSCdbErase([-iname=><name>,] <component ID>)**
  Discards the information associated with the specified component in the Customer Database. The component name is typically specified in canonical format (see `NMSEPIConvertCompId`). The command is synchronous as it blocks and waits for the reply from the CDB server.

  Example

  ```
  if ( NMSCdbErase(“EM TOTO FRUNI 52”) ) {
     ... # data is now erased
  ```

Real-Time Alarm Collection (RTAC)

The Real-Time Alarm Collection (RTAC) capabilities of EPI let you query the matching alarms to produce historical alarm reports. RTAC uses the rtaccol server to spool the alarms it retrieves from the GMDR server to workstation files, one file per day (based on the alarm’s time stamp). With the EPI RTAC interface, you can query the spooled alarms between two date-time boundaries. You can also specify filters for any alarm attribute. Specify these filters as GREP-style patterns. The matching alarms are retrieved and their attributes are provided in the same way as with the Alarm&Status specialized API interface.

Use RTAC Access commands n the following sequence:

1 Initialize an RTAC interface.

2 Start a query by specifying its date/time boundaries and attribute-value filters.

3 Fetch the matching alarms.

4 For each fetched alarm, extract the desired attributes and process them.

5 Drop the RTAC interface (usually not done).

The RTAC interface is synchronous because the interface can only perform one query at a time and the Fetch command is blocking.You can limit the amount of time and or the number of alarms the Fetch command scans to support a polling/round-robin form of multi-tasking.

The RTAC interface is also not remotable. It must be used on the same machine that stores (or NFS mounts) the RTAC spool files. The spool files are located using the `RTAC.cfg` configuration file . For details, see Real time alarm collection tool (rtaccol) in 241-6001-310 *Preside MDM Server Reference Guide*.

RTAC Access commands

The following RTAC Access commands are available:

- **NMSRTACInit([-iname=><name>])**
  Initializes an RTAC interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultRTAC` to make it the default interface).

- **NMSRTACDrop(-iname=><name>)**
  Drops the specified RTAC interface (frees allocated resources).

- **NMSRTACStartQuery([-iname <name>,]**
  <start date/time>, <end date/time> ,
  [-filter=><attribute name>, <pattern>, ...])
  Initiates an RTAC query for the alarms within the specified start and end date/time. Only alarms whose attributes match the specified filters will be returned. The start time can be specified as a date (“`yyyy mm dd`”), date and time (“`yyyy mm dd hh mm ss`”), as the special values “`0000 00 00`” or “`ANY`” meaning the oldest available alarm, and as the special value “`NOW`” meaning the current (workstation) date and time. Similarly, the end date/time can be specified as a date (“`yyyy mm dd`”), date and time (“`yyyy mm dd hh mm ss`”), as the special values “`9999 99 99`” or “`ANY`” meaning the latest alarm available, or as the special value “`NOW`”, meaning the current (workstation) date and time.
  The filter attribute names are the same provided through the Alarm&Status API. For details, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The filter values are specified as GREP style patterns for target values similar to the ones output by the

Alarm&Status API. Multiple attribute-pattern filters can be specified. Filters on the same attribute are ORed together and filters on different attributes are ANDed together (similar to the API rules).

Examples

```
NMSRTACInit();
if ( NMSRTACStartQuery("ANY", "NOW", \
              -filter=>  event,    "set", \
                         severity, "critical", \
                         severity, "major") ) {
   ... # ready to fetch all critical/major alarms up
   ... # to now

or,

NMSRTACInit();
NMSEPIConvertTime(-api, -offset=>-7200);
if ( NMSRTACStartQuery("$NMSEPI_OUTPUT_TIME",
              "NOW") ) {
   ... # ready to fetch alarms for the last two hours
```

To initiate a new query, simply invoke `NMDRTACStartQuery` again.

- **NMSRTACFetchNextAlarm([-iname=><name>.] [-out,]**
  [-terse|-normal|-full,]
  [-var=><array name>,]
  [-timeout=><timeout>,]
  [-maxrecs=><max records>])

  This command fetches the next matching alarm according to the criteria specified by `NMSRTACStartQuery` and `NMSRTACAddFilter`. The matching alarms are returned similarly to the `NMSGMDRAPIRecvAlarm` command. The following global variables are set;

  `NMSEPI_RECORD_TYPE:`
  is the API record type (always RESPONSE in this context).

  `NMSEPI_ALARM_SEVERITY:`
  is the severity of the alarm.

  `NMSEPI_ALARM_EVENT`
  is the alarm event.

NMSEPI_ALARM_FAULT
is the fault code.

NMSEPI_ALARM_TIME
is the alarm time.

NMSEPI_ALARM_COMPID
is the component ID.

NMSEPI_ALARM_OPDATA
is the operator data.

If you specify `-out` , the major attributes are also returned as a string result. If you specify `-terse`, `-normal`, or `-full`, then the `NMSEPI_ALARM_DISPLAY` global variable is set to the corresponding display format (similarly to `NMSGMDRAPIFormatAlarm` and `NMSRTACFormatAlarm`) and returned as a string result if `-out` is also specified. If you specify an associative array name with `-var`, the array is filled with all the non-empty attributes of the alarm just like for `NMSGMDRAPIRecvAlarm`.

```
if ( NMSRTACStartQuery("ANY", "NOW", \
             -filter=>  event,    "set", \
                        severity, "critical", \
                        severity, "major") ) {
   while ( NMSRTACFetchNextAlarm("-full",
             -var=>alarm) ) {
        if { $alarm{compCriticality} > 60 ) {
             print "$NMSEPI_ALARM_DISPLAY";
        }
   }
```

If you use the `-timeout` or `-maxrecs` options, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum

number of records tested occurs, the command returns the exit code `5` (timeout) as the `NMSEPI_RESULT` variable. This lets you "poll" the RTAC interface for a while and round-robin to other tasks.

You can extract the fetched alarm's contents with the commands `NMSRTACFormatAlarm`, `NMSRTACGetAlarm`, `NMSRTACGetFirstAttribute`, `NMSRTACGetNextAttribute`, and `NMSRTACFindNextAttribute` commands.

- **NMSRTACFormatAlarm([-iname=><name>,] [-out,] terse|normal|full)**

  This command produces the last fetched alarm in the specified display format. The output is provided as the `NMSEPI_ALARM_DISPLAY` global variable and as a string result if the `-out` option is specified.

  Example

```
if ( NMSRTACStartQuery("ANY", "NOW", \
                -filter=>  event,    "set", \
                           severity, "critical", \
                           severity, "major") ) {
    while ( NMSRTACFetchNextAlarm(-var=>alarm) ) {
        if ( $alarm{compCriticality} > 60 ) {
               NMSRTACFormatAlarm("full");
               print "$NMSEPI_ALARM_DISPLAY";
        }
    }
}
```

- **NMSRTACGetAlarm([-iname <name>,] [-out,] [-var=><array name>])**

  This command resets the attribute scan to the beginning of the attribute list (for `NMSRTACGetNextAttribute` and `NMSRTACFindNextAttribute`) and produces the last fetched alarm into the named associative array variable (if `-var` is used) and/or as a string result (if `-out` is used).

- **NMSRTACGetFirstAttribute([-iname=><name>,] [-out])**

  This extracts the first attribute of the last fetched alarm. The attribute is provided similarly to the `NMSAPIGetNextAttribute` command.

  `NMSEPI_FIELD_LABEL`
  is the API field type (always "`_attr:`" in this context).

NMSEPI_FIELD_NAME
is the API attribute name

NMSEPI_FIELD_TYPE
is the API attribute type element (always "S" in this context).

NMSEPI_FIELD_VALUE
is the API attribute value.

Multiple-line attribute block values are returned as a single multiple-line value.

If -out is used, the field is returned as a string result in an API-like format. The return error code is in NMSEPI_RESULT (6 indicates there are no more fields).

- **NMSRTACGetNextAttribute([-iname <name>,] [-out])**
  Similar to NMSRTACGetFirstAlarm but extracts the next attribute in the list.

  Example
  ```
  NMSRTACGetAlarm();
  ...
  while ( NMSRTACGetNextAttribute() ) {
     if (   "$NMSEPI_FIELD_LABEL" eq "_attr:" \
         && "$NMSEPI_FIELD_NAME"  eq "time" ) {
        print "Time is: $NMSEPI_FIELD_VALUE";
  ```

- **NMSRTACFindNextAttribute([-iname=><name>,] [-out,] <attribute name>)**
  Finds the next named (NRS type name or long name/title) attribute in the attribute list and returns it similarly to NMSRTACGetFirstAttribute.

Network Reporting System

The Network Reporting System (NRS) capabilities of EPI let you create device configuration reports on the data stored in the NRS database and its schema (RDF files). You build the NRS query by identifying the node configuration files to scan (by name, pattern, and naming discipline such as keyed and dated)and the component types to extract configuration data from. The configuration data is returned to the EPI script in various ways (for

example, environment variable, associative arrays, and standard output). The data values correspond to the way NRS reports currently work. The EPI NRS capabilities also add value by providing program access to the following items:

- NRS schema contents (the RDF files)
- the automatic construction of the component name
- the ability to specify Passport component and attribute types by name and hierarchical path, rather than by numerical IDs.

EPI does not provide a means of populating this database. You must use the NRS utilities for this (some examples include nrspop, pnrspop, and sisauto). For more information on the NRS database and its use, see 241-6001-022 *Preside MDM Network Reporting System User Guide*.

Use the following sequence for NRS Access commands:

1 Initialize an NRS interface.

2 Start a query by specifying the target node configurations and the component types to be reported, and/or

3 Load in and interrogate the NRS schema to help create the report (optional).

4 Fetch the matching configuration components. Components are extracted in depth-first order with no garanteed ordering at peer level (as with other NRS reporting mechanisms).

5 For each fetched component, extract the desired attributes and process them.

6 Drop the NRS interface (usually not done).

The NRS interface is synchronous because an interface can only perform one query at a time and the Fetch command is blocking. However, you can limit the amount of time and the number of components the Fetch command scans to support a polling/round-robin form of multi-tasking.

You cannot use the NRS interface remotely. Use the NRS interface on the same machine that stores (or NFS mounts) the NRS database. The database and schema are located using the `NRS.cfg` configuration file. For details, see 241-6001-022 *Preside MDM Network Reporting System User Guide*.

- **NMSNRSInit([-iname=><name>])**
  Initializes an NRS interface. If you use multiple interfaces at the same time, specify a name in the command invocations for this interface. This name must be indicated with the `-iname` option on all commands targeted at this interface (or use `NMSEPIDefaultNRS` to make it the default interface).

- **NMSNRSDrop(-iname=><name>)**
  Drops the specified NRS interface (frees allocated resources).

- **NMSNRSStartQuery([-iname=><name>,]**
  [[-include=>] <comp. type>[*|+|^], ...]
  [-exclude=><comp. type>[*|+|^], ...])
  Initiates an NRS query. The query reports the specified component types. Types listed after the `-exclude` option are not reported. By default, no types are reported. Component types are specified as a quoted string consisting of

  ```
  [<device type>/]<component>[*|+|^]
  ```

  where `<device type>` is a recognized NRS device type (`ppc` for Carrier versions of Passports, `ppe` for Enterprise versions, `dpn` for DPN equipment). If no device type is specified, the default type for the installation is taken from the `NRS.cfg` file. The `<component>` specification can be a name for DPN components, a numerical component ID for Passport components, or a name (the full component type name, for example, `FrameRelayUni`, or the prompt, for example, `FrUni`) for Passport components.

  Note: The Passport component names are not unique since there can be multiple Passport components with the same name or prompt. Passport component types are uniquely identified by their numeric component ID. Including or excluding component types to the report by name includes or excludes all the possible matches. This may result in unwanted components being reported.

  Passport components can also be identified as the full hierarchy of full names and/or abbreviations. For example, to include the `ServiceParameter` component of a Frame Relay UNI Dlci, the following component type can be specified: `ppc/EM-FrUni-Dlci-Sp`. In that case, only the specified Sp subcomponent (FrUni-Dlci-Sp) will be returned. `Sp` components of other hierarchies (for example, FrNni-Dlci-Sp) will be ignored.

The names are not case sensitive. The schema, RDF files, to interpret the names are located as specified in the `NRS.cfg` file.

If the component type specification uses an asterix (`*`) as a suffix, all the component's possible subcomponents (recursively) are included or excluded, as specified. If you use a plus sign (+), only its immediate subcomponents are included or excluded. If you use a caret (^) as a suffix, then all the component's possible parent components are included or excluded. If the modifier is applied to a full path Passport component specification, then only the related components of the specified full path are added. In non-path specifications, all possible parents are included or excluded. You can combine multiple suffixes. It is possible to include whole sub-hierarchies of components then exclude the subcomponents you don't need by using the `-exclude` option. More component types can be include/excluded with the `NMSNRSReportCompType` command. Specify the `-include` option if you want to include more components after the specification of excluded ones.
To start a new NRS query, call `NMSNRSStartQuery` again.

Example(1)

```
NMSNRSInit();
if ( NMSNRSStartQuery("ppc/2", "ppc/FrameRelayUni*") )
{
    ... # ready to specify the source files for an NRS
    ... # report on the (Carrier) Passport module and
    ... # Frame Relay components (including all its
    ... # subcomponents)
```

Example(2)

```
NMSNRSInit();
if ( NMSNRSStartQuery("ppc/EM-AtmIf-Vcc-Vcd^") ) {
    ... # ready to specify the source files for an NRS
    ... # report on the (Carrier) Atm Virtual Connection
    ... # Description and its indicated parents (Vcc,
    ... # AtmIf and EM).
```

- **NMSNRSReportCompType([-iname=><name>,]**
  [[-include=>] <comp. type>[*|+|^], ...]
  [-exclude=><comp. type>[*|+|^], ...]
  | -list [, -out])

  Includes or excludes additional component types to the report. See

`NMSNRSStartQuery` for the specification of the types. `NMSNRSReportCompType` can be called multiple times to include and exclude component types.

This command can also be used to list, with the `-list` option, the component types currently included. These are listed, one per line, as the value of the `NMSEPI_OUTPUT_TEXT` environment variable. If `-out` is specified, they are also returned as a string result.

Note: The reported component types can be manipulated in the middle of a report by invoking the `NMSNRSReportCompType` command as appropriate.

Example

```
if ( NMSNRSStartQuery(“ppc/2”, “ppc/FrameRelayUni*”) )
{
    NMSNRSReportCompType(-exclude=>”ppc/Signalling”, \
          “ppc/DataLinkConnectionIdentifier*”);
    ... # Same as previous example except that this time
    ... # the Signalling and DLCI subcomponents (and
    ... # all its subcomponents for the latter) are to
    ... # to be excluded from the report
```

- **NMSNRSAddSource([-iname=><name>]**
  -file=><file name>
  | -named=><device>, <module>, <name>
  | -keyed=><device>, <module>, <key>
  | -dated=><device>, <module>, <date>
  | -latest=><device>, <module>
  [, <from date/time>]
  | -list [, -out])

  This command specifies which NRS data files to report on. The files are located at the path specified in the `NRS.cfg` file. The files are expected

to already be there. For details about how to populate the NRS database, see 241-6001-022 *Preside MDM Network Reporting System User Guide*. The file(s) can be specified in a number of ways;

`-file`	Explicitly names the NRS data file to report on (with or without a full path -- the NRS database path as configured in `NRS.cfg` will be used if not specified).
`-named`	Includes NRS data file(s) matching the specified GREP patterns; `<device>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<module>`, the module name, and `<name>`, the configuration file name.
`-keyed`	Includes NRS data file(s) matching the specified GREP patterns; `<device>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<module>`, the module name, and `<key>`, the configuration file key. Keyed configuration file names have a fixed prefix (the key) followed by a variable two-digit counter. `<key>` only matches the key prefix. For a matching key, the file with the highest two-digit suffix is selected.

-dated	Includes NRS data file(s) matching the specified GREP patterns; <device>, the device type (for example, ppc, ppe, and dpn), <module>, the module name, and <date>, the configuration file date (six digits, not a pattern). Dated configuration file names have a fixed six-digit prefix (the date) followed by a variable two digit counter. <date> only matches the date prefix. For Passports, <date> matches the activation date of the NRS data file name. For both Passport and DPN files, the highest dated file up to the specified date is accepted (and the one with the highest two-digit counter suffix).
-latest	Includes NRS data file(s) matching the specified GREP patterns; <device>, the device type (for example, ppc, ppe, dpn), and <module>, the module name. If you specify <fromDateTime> as "YYMMDD", "YYYY MM DD", or "YYYY MM DD HH MM SS", only the NRS data files from that date/time forward (UNIX file modifitation time-stamp) are considered. This option is useful for creating incremental reports based on the latest NRS population or the last report invocation. For each matching module, the matching file with the highest file system date is selected (the most recently populated file).

In all cases where multiple files match the patterns provided in the `NMSNRSAddSource` call, only one file per module is selected, the highest one in alphanumerical order. For Passport, this also means the one with the highest version counter, the three-digit file suffix. Multiple calls to `NMSNRSAddSource` can select multiple files for the same module.

This command can also be used to list, with the `-list` option, the files currently included in the report. They are listed, one per line, with full path, as the value of the `NMSEPI_OUTPUT_TEXT` environment variable. If you specify `-out`, they are also returned as a string result.

Example(1)

This example and the following assume a sample NRS database containing the following files:

```
dpn.R78.4078.R7872.970709.data
dpn.R78.4078.R7888.970715.data
ppc.NODER16.2105.NEWCARD,full,012.001105.data
ppc.NODER16.2105.NEWCARD05,full,001.001102.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
ppc.NODER17.2106.newconf,full,030.010210.data
ppc.NODER17.2106.demo,full,003.010123.data
ppc.EASTOTT.2100.lab,full,005.010123.data
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
ppe.SOUTHRICH.2102.JF,full,003.010124.data
ppe.SOUTHRTP.2113.voice,full,044.010124.data
```

```
if ( NMSNRSStartQuery("dpn/PE^") ) {
   NMSNRSAddSource(-file=>\
          "dpn.R78.4078.R7872.970709.data");
   ... # ready to fetch module and PE components from
   ... # the specified NRS data file
```

Given the preceding sample database, this call selects the following file:

```
dpn.R78.4078.R7872.970709.data
```

Example(2)

```
if ( NMSNRSStartQuery("ppc/Card") ) {
   NMSNRSAddSource(-name=>"ppc", ".*", "NEWCARD.*");
   ... # ready to fetch card components from all
   ... # node configurations whose name start with
```

```
    ... # NEWCARD
```

Given the sample database, this call selects the following file:

```
    ppc.NODER16.2105.NEWCARD,full,012.001105.data
            (the "latest" of the two matching files for NODER16)
    ppc.NODER12.2101.NEWCARD09,full,001.001102.data
```

Example(3)

```
if ( NMSNRSStartQuery("ppc/Card") ) {
    NMSNRSAddSource(-keyed=>"ppc", ".*", "NEWCARD");
    ... # similar as above but with strict KEY syntax
```

Given the sample database, this call selects the following file:

```
    ppc.NODER16.2105.NEWCARD05,full,001.001102.data
    ppc.NODER12.2101.NEWCARD09,full,001.001102.data
```

(The other NODER16 file was not selected as its name does not match the syntax of a keyed file)

Example(4)

```
if ( NMSNRSStartQuery("ppc/Card") ) {
    NMSNRSAddSource(-dated=>"ppc", ".*", "010211");
    ... # as above but this time for all node
    ... # configurations dated before or for February
    ... # 11th 2001
```

Given the sample database, this call selects the following file:

```
    ppc.NODER16.2105.NEWCARD,full,012.001105.data
    ppc.NODER12.2101.NEWCARD09,full,001.001102.data
    ppc.NODER17.2106.newconf,full,030.010210.data
    ppc.EASTOTT.2100.lab,full,011.010124.data
    ppc.EASTMTL.2109.demo,full,031.010104.data
```

(All configurations up to 010211 for PPC nodes are taken.)

Example(5)

```
if ( NMSNRSStartQuery("ppc/Software", "ppc/Software")
) {
    NMSNRSAddSource(-latest=>"ppc", "EAST.*");
    NMSNRSAddSource(-latest=>"ppe", "SOUTH.*");
    ... # ready to fetch Software components from the
    ... # latest configuration of the nodes whose names
    ... # start with EAST or SOUTH
```

Given the sample database, this call selects the following file:

```
    ppc.EASTOTT.2100.lab,full,011.010124.data
```

```
ppc.EASTMTL.2109.demo,full,031.010104.data
ppe.SOUTHRICH.2102.JF,full,003.010124.data
ppe.SOUTHRTP.2113.voice,full,044.010124.data
```

(Assuming the UNIX file data matches the ordering of the activation dates in the file names.)

- **NMSNRSFetchNextComponent([-iname=><name>,] [-out,] [-var=><array name>, [-byname,]] [-skip=><level>,] [-stop=><level>,] [-timeout=><timeout>,] [-maxrecs=><maxrecs>]) [-marker <seek marker>])**

Fetches the next matching component (according to the component types specified in `NMSNRSStartQuery` and `NMSNRSReportCompType`) from the selected (through `NMSNRSAddSource`) NRS data files. The matching component information is available through the following global variables:

`NMSEPI_NRS_COMPID`
is the full Component ID of the fetch component (the component is specified in mixed case, with space separators, for example, "`EM NODER16 FrUni 132 Dlci 206 Sp $`", and can be manipulated with `NMSEPIConvertCompId`).

`NMSEPI_NRS_COMPTYPE`
is the component type of the fetched component (the component type is specified as `<device>/<type>` where `<device>` is the NRS device type (`ppc`, `ppe`, or `dpn`), and `<type>` is the component type (a name for DPN, a numerical component ID for Passport, for example, `dpn/PE` or `ppe/8664`).

`NMSEPI_NRS_COMPTITLE`
is the (long) type name of the fetched component, for example, `ServiceParameterProv` for the component identified above.

`NMSEPI_NRS_COMPABBREV`
is the short type name of the fetched component (for Passport's, it is the prompt), for example `Sp` for the component identified above.

`NMSEPI_NRS_VALUE`

is the instance value of the fetched component (the value of its most specific component ID category/value pair, for example, "$" for the component identified above).

NMSEPI_NRS_LEVEL
is the hierarhcy level of the fetched component (starting at 0 for the module level, for example, 3 for the component identified above). Note that matching components are returned in depth-first order.

NMSEPI_NRS_FILE_PATH
is the NRS data file from which the matching component was fetched.

If you specify the `-out` option, the first four items above are also output, one per line, to the standard output stream. If you use the `-var` optionto name an associative array, the array dills with the NRS component description. The array entry index is the attribute type, as specified in NRS, names for DPN and the special values "`OAM`", "`_COMPONENT`", "`_HIERARCHY_LEVEL`", and a numerical attribute ID for Passport components (for example, `16567` for the `Frame Relay Sp committedInformationRate` attribute). If you specify the `-byname` option, the index is as indicated except for the special values, which are then reported as "`Ownership`", "`Component`" and "`Hierarchy_level`" respectively, and for Passport components for which the full attribute name is used instead (for example, "`committedInformationRate`", not "`cir`"). The entry value is the corresponding attribute value. EPI also includes special entries with `_EPI_COMPID`, `_EPI_TITLE`, and `_EPI_ABBREV` index containing the information corresponding to the `NMSEPI_NRS_COMPID`, `EPI_NRS_COMPTITLE`, and `EPI_COMPABBREV` variables above.

The individual attributes of the fetched component can also be extracted one at a time using the `NMSNRSGetFirstAttribute`, `NMSNRSGetNextAttribute`, and `NMSNRSFindAttribute` commands.

If you use the `-marker` option, the specified parameter must be a component marker previously saved from a call to `NMSNRSGetComponentMarker`. This repositions the NRS file scanning to the specified marker before fetching the next matching component. If

the same component type specifications are used, the same component whose marker is saved is returned. See the description of `NMSNRSGetComponentMarker` for more information. If the specified marker is an empty string, the option is ignored (as if it was not specified).

A marker may refer to another file than the one being scanned. In this case, the marked file is pushed onto the interface and the current file is re-scanned from the top when the marked file is fully processed.

If you use the `-skip` option, to specify a hierarchy level, the command returns the next matching component of a hierachy level that is smaller or equal to the one given. Other matching components of a higher level are ignored. This method restricts the search by skipping over components that are found to be undesireable (for example, if a Service Parameter (level 3) component attribute of a Passport Frame Relay DLCI component (level 2) does not meet the necessary criteria, the next call to `NMSCmdFetchNextComponent` can specify `-skip=>2` which skips to the next DLCI component, ignoring any intervening matches. NRS components are delivered in depth-first order.

Note: Specifying `-skip=>0` is an efficient way of skipping over an entire module (configuration file) since EPI does not to try to match the left over components. The command tries to match components in the next module-configuration specified with `NMSNRSAddSource`. Specifying `-skip=>"-1"` has the same effect as not specifying the option at all.

If you use the `-stop` option, the next matching component is returned as normal. In addition, if a non-matching component of the specified level or a lower value is found, the command returns with an empty component (all variables are empty except for `NMSEPI_NRS_LEVEL`). As a result, you receive information when the member of a component sub-tree is scanned and another sub-tree of the same level is about to be scanned (the new sub-tree matches, then it is returned as normal, else the empty component is returned as a form of component terminator). For example, assuming `FrUnis` (level 1) are being extracted with `-stop 1` specified as soon as the first one is found, all matching `FrUnis` will be returned as normal and an empty component is returned once a `Vs`

components (also at level 1 but not requested by the query) is found. This technique allows one to maintain a state machine that knows when not to expect more sub-components of a specific sub-tree.

If you set the `-timeout` or `-maxrecs` options, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum number of records tested occurs, the command returns the exit code `5` (timeout) as the `NMSEPI_RESULT` variable. This lets you "poll" the NRS interface for a while and round-robin to other tasks.

Example(1)

```
# looking for FrUnis with no running LMIs
...
NMSNRSStartQuery("ppc/FrUni+");
...
$skip = -1;
while ( NMSNRSFetchNextComponent(-skip=>$skip, \
          -var=>comp, "-byname"} ) {
   $skip = -1;
   if (   $NMSEPI_NRS_COMPABBREV eq "Lmi" \
       && $comp(procedures) eq "none" } {
          print \
          "$NMSEPI_NRS_COMPID has no active Lmi";
   } else {
          # skip to the next FrUni
          $skip = $NMSEPI_NRS_LEVEL - 1;
   }
}
```

Example(2)

```
...
NMSNRSStartQuery("ppc/ServiceParametersProv");
...
NMSNRSFetchNextComponent(-var=>comp, "-byname");
```

can produce the following values for the `comp` array:

```
  $comp{_EPI_COMPID} = \
         "EM NODER16 FrUni 132 Dlci 206 Sp"
  $comp{_EPI_TITLE} = ServiceParametersProv
  $comp{_EPI_ABBREV} = Sp
  $comp{Component} = ppe/8664
```

```
$comp{Hierarchy_level} = 3
$comp{Ownership} = OWNER_IWS
$comp{EM} = NODER16
$comp{FrameRelayUni} = 132
$comp{DataLinkConnectionIdentifier} = 206
$comp{ServiceParametersProv} = $
$comp{committedInformationRate} = 64000
$comp{measurementInterval} = 0
$comp{committedBurstSize} = 64000
$comp{rateEnforcement} = on
$comp{rateAdaptation} = off
$comp{excessBurstSize} = 0
$comp{maximumFrameSize} = 2100
$comp{accounting} = on
$comp{updateBCI} = off
$comp{raSensitivity} = 7
```

without the `-byname` option, the output is:

```
$comp{_EPI_COMPID} = \
       "EM NODER16 FrUni 132 Dlci 206 Sp $"
$comp{_EPI_TITLE} = ServiceParametersProv
$comp{_EPI_ABBREV} = Sp
$comp{_COMPONENT} = ppe/8664
$comp{_HIERARCHY_LEVEL} = 3
$comp{OAM} = OWNER_IWS
$comp{_2} = NODER16
$comp{_279} = 132
$comp{_302} = 206
$comp{_8664} = $
$comp{8668} = on
$comp{8670} = 64000
$comp{8669} = 64000
$comp{5997} = off
$comp{8671} = 0
$comp{8672} = 0
$comp{8673} = off
$comp{8674} = 7
$comp{8675} = on
$comp{8667} = 2100
```

- **NMSNRSGetComponentMarker [-iname <name>] [-out]**
  Extracts a marker for the current fetched component and returns it as the `NMSEPI_NRS_COMP_MARKER` environment variable or on standard

output if `-out` is used. The marker may be saved and used later as an argument to `NMSNRSFetchNextComponent's -marker` option to rescan a component structure (not the whole file) which works around the NRS peer component type that is not specified (all instances of a subcomponent type X are together, but it is not specified if instances of type X appear before or after those of its peer type Y). For example, an NRS interface may be used to scan for Frame Relay interfaces and their DNA sub-components. When an FrUni is found, its marker is saved and the interface is used to locate its Dna. Then the same (requires one to manipulate the selected types with `NMSNRSReportCompType`) or another NRS interface can be used to scan for the FrUni's Dlci subcomponents. The marker may be exchanged between properly initialized NRS interfaces. markers may also refer to different files. If the marker cannot be computed, an empty string is returned.

Example

```
# Assume two NRS interfaces (inames fruni and dlci)
# configured to scan for FrUnis/Dnas and Dlcis
# respectively
while (NMSNRSFetchNextComponent(-iname=>fruni, \
          -var=>comp) {
    if ( $NMSEPI_NRS_COMP_ABBREV eq "FrUni" ) {
          # ... process FrUni information...
          NMSNRSGetComponentMarker();
          $marker = "$NMSEPI_NRS_COMP_MARKER";
    } else if ( $NMSEPI_NRS_COMP_ABBREV eq "Dna" ) {
          # ... process DNA information ...
          stopAt = -1;
          while ( NMSNRSFetchNextComponent( \
                      -iname=>dlci, -stop=>$stopAt, \
                      -marker=>"$marker") ) {
             $marker = "";
             $stopAt = 1;
             # ... process the DLCIs ...
          }
    }
}
```

- **NMSNRSGetComponent([-iname=><name>,] [-out,]**
  [-var=><array name> [, -byname]])

  Resets the fetched component's attribute list for the

`NMSNRSGetFirstAttribute`, `NMSNRSGetNextAttribute`, and `NMSNRSFindAttribute` commands and returns the description of the component similarly to `NMSNRSFetchNextComponent`.

- **NMSNRSGetFirstAttribute([-iname=><name>,] [-out,] [-var=><array name>])**

  Extracts the first attribute from the fetched component. The following global variables are set with the description of the attribute:

  `NMSEPI_FIELD_LABEL`
  is the attribute's name (for Passport, it is the attribute's full name, not its prompt, for example, "`Percent heap`" for DPN, "`committedInformationRate`" for Passport).

  `NMSEPI_FIELD_NAME`
  is the attribute's type; a string for DPN, a numeric attribute ID for Passport (for example, "`LOADPETYPE`" for DPN, `8669` for Passport).

  `NMSEPI_FIELD_TYPE`
  is the NRS type of the attribute; `BIT_STRING`, `BOOLEAN`, `DNA`, `HEXADECIMAL`, `INVISIBLE`, `LISTINDEX`, `NOKEY`, `NUMERIC`, or `STRING`.

  `NMSEPI_FIELD_VALUE`
  is the attribute's value.

  If `-out` is specified, the values are also returned, one line in the same order as above, as a string result.
  If `-var` is used to name an associative array, that array is filled with the attribute's description with entries with the following indicies:

  `name`
  the attribute's type (same as `NMSEPI_FIELD_NAME` above).

  `value`
  the attribute's value (same as `NMSEPI_FIELD_VALUE` above).

  `type`
  the attribute's NRS type (same as `NMSEPI_FIELD_TYPE` above).

  `title`

the attribute's full name (same as `NMEPI_FIELD_LABEL` above).

`abbrev`
for DPN, it is the same as the `title` entry, for Passport it is the attribute's prompt.

`width`
the attribute's maximum width in NRS.

- **NMSNRSGetFirstAttribute([-iname=><name>,] [-out,] [-var=><array name>])**
  Extracts the next attribute from the fetched component. The attribute information is returned similarly to `NMSNRSGetFirstAttribute`.

- **NMSNRSFindAttribute([-iname=><name>] [-out,] [-var=><array name>,] <name>)**
  Extracts the named attribute from the fetched component. The search is case-insensitive. For Passport, both the full attribute name and the prompt can be used. The attribute information is returned similarly to `NMSNRSGetFirstAttribute`.

  Example

```
# looking for FrUnis with no running LMIs (as above)
...
NMSNRSStartQuery("ppc/EM-FrUni-Lmi");
...
$skip = -1;
while ( NMSNRSFetchNextComponent(-skip=>$skip) ) {
    $skip = -1;
    NMSNRSFindAttribute("procedures");
    if ( $NMSEPI_FIELD_VALUE eq "none" ) {
           print \
           "$NMSEPI_NRS_COMPID has no active Lmi\n";
    } else {
           # skip to the next FrUni
           $skip = $NMSEPI_NRS_LEVEL - 1;
    }
}
```

- **NMSNRSGetComponentSchema([-iname=><name>,] [-out,] [-var=><array name>, [-byname,]]**

[<comp type>[*|+|^]])

Use this command to examine the component schema for the current fetched component (from `NMSNRSFetchNextComponent`) when no `<comp type>` is specified. You can also use it to examine the NRS schema (RDFs) in general when `<comp type>` is specified in a similar manner to the `NMSNRSStartQuery` command. Use the `*`, `+`, or `^` prefix to automatically load the RDFs for the subcomponents and parents of the specified component respectively. (Note that components loaded with this command are not necessarily reported. Use `NMSNRSStartQuery` or `NMSNRSReportCompType` for this).

Note: For Passport, `<comp type>` can also be specified as the component's numerical ID, name, prompt or full name/prompt path. Passport component names and prompts are not unique since there can be multiple Passport components with the same name or prompt (but different numerical IDs). It is not determined which matching component will be returned by `NMSNRSGetComponentSchema` in that case.

The following global variables are set by this command:

`NMSEPI_NRS_SCHEMA_NAME`
is the component's type specified as `<device>/<comp>` where `<device>` is the device type (`ppc`, `ppe`, or `dpn`), and `<comp>` is the component type (a name for DPN, a numerical ID for Passport, for example, `dpn/UTP` for DPN, `ppc/302` for Passport).

`NMSEPI_NRS_SCHEMA_TITLE`
is the component's name (a name for DPN, the full component name, for Passport, for example, "`UTP`" for DPN, "`FrameRelayUni`" for Passport).

`NMSEPI_NRS_SCHEMA_ABBREV`
is the component's abbreviation (a name for DPN, the prompt for Passport, for example, "`UTP`" for DPN, "`FrUni`" for Passport).

`NMSEPI_NRS_SCHEMA_OWNERS`
is the list of the component's direct parent component types. The types are on a single line, space separated, and specified as `<device>/<comp>` similarly to `NMSEPI_NRS_SCHEMA_NAME` above. For example, the owners for `dpn/UTP` is "`dpn/PO`", and for `ppc/VritualFramer`

they are “ppc/9200 ppc/1543 ppc/279 ppc/3742“.

NMSEPI_NRS_SCHEMA_SUBORDINATES
is the list of the component’s direct sub-component types. The types are on a single line, space separated, and specified as <device>/<comp> similarly to NMSEPI_NRS_SCHEMA_NAME above. For example, for dpn/UTP the subordinates are “dpn/UTPLINK dpn/UTPLINK_EXT dpn/MI8_CUS_MODEM_PROF_ENV dpn/UTPDIALUP dpn/UTPPASSWD_ENV” , and for ppc/DataLinkConnectionIdentifier they are “ppc/8664 ppc/219 ppc/16565“.

NMSEPI_NRS_SCHEMA_ATTRS
is the list of the component’s attribute types. The types are on a single line, space separated, and specified as names for DPN and as numerical IDs for Passport. For example, for dpn/UTP, they attributes are “_COMPONENT _HIERARCHY_LEVEL _PM _PE _PI _PO _UTP OAM“, and for ppc/FrameRelayUni they are “_COMPONENT _HIERARCHY_LEVEL _2 _279 OAM 96 3177 3175 9332“.

If you specify -out, the component type, (full) name, abbreviation (prompt), list of direct parent component types, list of direct subcomponent types, and list of attributes are returned as a string result one per line (the last two lists are on a line each, with space separators between the types).

If you use -var to name an associative array, the command fills two arrays with the schema information. The <array name> array contains the following entries indicated by their index:

name
is the component’s type (same as NMSEPI_NRS_SCHEMA_NAME above).

title
is the component’s name (same as NMSEPI_NRS_SCHEMA_TITLE above).

abbrev
is the component’s abbreviation (same as NMSEPI_NRS_SCHEMA_ABBREV above).

owners
is the component's list of direct parent types (same as NMSEPI_NRS_SCHEMA_OWNERS above).

subordinates
is the component's list of direct subcomponent types (same as NMSEPI_NRS_SCHEMA_SUBORDINATES above).

attrs
is the component's attribute types (same as NMSEPI_NRS_SCHEMA_ATTRS above).

The <array name>_fields array contains the following entries for each attribute types supported by the component (where <field type> is the attribute name for DPN and the attribute numeric ID for Passport or the attributes full name/title if -byname is specified).

<field type>,**title**
is the indexed field's name (a string for DPN, the full attribute name for Passport).

<field type>,**name**
is the indexed field NRS attribute type name.

<field type>,**type**
is the indexed field NRS type (see the description of NMSEPI_FIELD_TYPE for NMSNRSGetFirstAttribute).

<field type>,**width**
is the indexed field maximum NRS width.

<field type>,**abbrev**
is the indexed field abbreviation (same as the title for DPN, the attribute prompt for Passport).

<field type>,**group**
is the indexed field attribute group type.

Example(1)

```
# Lists (by full name) the attributes of the Passport
# Lp component
NMSNRSGetComponentSchema(-var=>comps, \
          "ppc/LogicalProcessor");
print "$NMSEPI_NRS_SCHEMA_TITLE\n";
# print its attributes by name
foreach $i ( split / /, $NMSEPI_NRS_SCHEMA_ATTRS ) {
    $ii = "$i,title";
    print " $comps_fields{$ii}\n";
}
```

Example(2)

```
NMSNRSInit();
NMSNRSGetComponentSchema(-var=>comp, "dpn/UTP");
```

can produce the following values for the comp array:

```
$comp{name) = dpn/UTP
$comp{title) = UTP
$comp{abbrev) = UTP
$comp{owners) = dpn/PO
$comp{subordinates) = "dpn/UTPLINK dpn/UTPLINK_EXT \
        dpn/MI8_CUS_MODEM_PROF_ENV dpn/UTPDIALUP \
        dpn/UTPPASSWD_ENV"
$comp{attrs) = "_COMPONENT _HIERARCHY_LEVEL _PM \
        _PE _PI _PO _UTP OAM"
```

and the following values for the comp_fields array:

```
$comp_fields{_COMPONENT,abbrev) = Component
$comp_fields{_COMPONENT,group) =
$comp_fields{_COMPONENT,name) = _COMPONENT
$comp_fields{_COMPONENT,title) = Component
$comp_fields{_COMPONENT,type) = STRING
$comp_fields{_COMPONENT,width) = 0
$comp_fields{OAM,abbrev) = Ownership
$comp_fields{OAM,group) =
$comp_fields{OAM,name) = OAM
$comp_fields{OAM,title) = Ownership
$comp_fields{OAM,type) = STRING
$comp_fields{OAM,width) = 10
$comp_fields{_HIERARCHY_LEVEL,abbrev) = \
        Hierarchy_level
```

```
$comp_fields{_HIERARCHY_LEVEL,group) =
$comp_fields{_HIERARCHY_LEVEL,name) = \
        _HIERARCHY_LEVEL
$comp_fields{_HIERARCHY_LEVEL,title) = \
        Hierarchy_level
$comp_fields{_HIERARCHY_LEVEL,type) = NUMERIC
$comp_fields{_HIERARCHY_LEVEL,width) = 2
$comp_fields{_PM,abbrev) = PM
$comp_fields{_PM,group) =
$comp_fields{_PM,name) = _PM
$comp_fields{_PM,title) = PM
$comp_fields{_PM,type) = STRING
$comp_fields{_PM,width) = 12
$comp_fields{_PE,abbrev) = PE
$comp_fields{_PE,group) =
$comp_fields{_PE,name) = _PE
$comp_fields{_PE,title) = PE
$comp_fields{_PE,type) = NUMERIC
$comp_fields{_PE,width) = 2
$comp_fields{_PI,abbrev) = PI
$comp_fields{_PI,group) =
$comp_fields{_PI,name) = _PI
$comp_fields{_PI,title) = PI
$comp_fields{_PI,type) = NUMERIC
$comp_fields{_PI,width) = 2
$comp_fields{_PO,abbrev) = PO
$comp_fields{_PO,group) =
$comp_fields{_PO,name) = _PO
$comp_fields{_PO,width) = 2
$comp_fields{_PO,type) = NUMERIC
$comp_fields{_PO,title) = PO
$comp_fields{_UTP,abbrev) = UTP
$comp_fields{_UTP,group) =
$comp_fields{_UTP,name) = _UTP
$comp_fields{_UTP,title) = UTP
$comp_fields{_UTP,type) = NOKEY
$comp_fields{_UTP,width) = 1
```

Example(3)

```
NMSNRSInit
NMSNRSGetComponentSchema(-var=>comp,
          “ppc/FrameRelayUni”);
```

can produce the following values for the comp array:

```
$comp{name) = ppc/279
$comp{title) = FrameRelayUni
$comp{abbrev) = FrUni
$comp{owners) = ppc/2
$comp{subordinates) = “ppc/161 ppc/16502 ppc/287 \
        ppc/10990 ppc/586 ppc/9314 ppc/302 \
        ppc/10649 ppc/8600 ppc/1768”
$comp{attrs) = “_COMPONENT _HIERARCHY_LEVEL _2 \
        _279 OAM 96 3177 3175 9332”
```

and the following values for the `comp_fields` array:

```
$comp_fields{_COMPONENT,abbrev} = Component
$comp_fields{_COMPONENT,name} = _COMPONENT
$comp_fields{_COMPONENT,title} = Component
$comp_fields{_COMPONENT,type} = STRING
$comp_fields{_COMPONENT,width} = 0
$comp_fields{OAM,abbrev} = Ownership
$comp_fields{OAM,name} = OAM
$comp_fields{OAM,title} = Ownership
$comp_fields{OAM,type} = STRING
$comp_fields{OAM,width} = 10
$comp_fields{_HIERARCHY_LEVEL,abbrev} = \
        Hierarchy_level
$comp_fields{_HIERARCHY_LEVEL,group} =
$comp_fields{_HIERARCHY_LEVEL,name} = \
        _HIERARCHY_LEVEL
$comp_fields{_HIERARCHY_LEVEL,title} = \
        Hierarchy_level
$comp_fields{_HIERARCHY_LEVEL,type} = NUMERIC
$comp_fields{_HIERARCHY_LEVEL,width} = 2
$comp_fields{_2,abbrev} = EM
$comp_fields{_2,group} =
$comp_fields{_2,name} = _2
$comp_fields{_2,title} = EM
$comp_fields{_2,type} = STRING
$comp_fields{_2,width} = 12
$comp_fields{_279,abbrev} = FrUni
$comp_fields{_279,group} =
$comp_fields{_279,name} = _279
$comp_fields{_279,title} = FrameRelayUni
$comp_fields{_279,type} = NUMERIC
$comp_fields{_279,width} = 5
```

```
$comp_fields{3175,abbrev} = ifI
$comp_fields{3175,group} = IfEntryProv
$comp_fields{3175,name} = 3175
$comp_fields{3175,title} = ifIndex
$comp_fields{3175,type} = NUMERIC
$comp_fields{3175,width} = 5
$comp_fields{3177,abbrev} = ifState
$comp_fields{3177,group} = IfEntryProv
$comp_fields{3177,name} = 3177
$comp_fields{3177,title} = ifAdminStatus
$comp_fields{3177,type} = STRING
$comp_fields{3177,width} = 7
$comp_fields{96,abbrev} = cid
$comp_fields{96,group} = CustomerIdentifierData
$comp_fields{96,name} = 96
$comp_fields{96,title} = customerIdentifier
$comp_fields{96,type} = STRING
$comp_fields{96,width} = 4
$comp_fields{9332,abbrev} = numberOfEmissionQs
$comp_fields{9332,group} = EmissionPriorityQs
$comp_fields{9332,name} = 9332
$comp_fields{9332,title} = numberOfEmissionQs
$comp_fields{9332,type} = STRING
$comp_fields{9332,width} = 1
```

If -byname is specified, the fields would report as follows:

```
$comp_fields{Component,abbrev} = Component
$comp_fields{Component,group} =
$comp_fields{Component,name} = _COMPONENT
$comp_fields{Component,title} = Component
$comp_fields{Component,type} = STRING
$comp_fields{Component,width} = 0
$comp_fields{OAM,abbrev} = Ownership
$comp_fields{Ownership,name} = OAM
$comp_fields{Ownership,group} =
$comp_fields{Ownership,title} = Ownership
$comp_fields{Ownership,type} = STRING
$comp_fields{Ownership,width} = 10
$comp_fields{Hierarchy_level,abbrev} = \
        Hierarchy_level
$comp_fields{Hierarchy_level,group} =
$comp_fields{Hierarchy_level,name} = \
```

```
        _HIERARCHY_LEVEL
$comp_fields{Hierarchy_level,title} = \
       Hierarchy_level
$comp_fields{Hierarchy_level,type} = NUMERIC
$comp_fields{Hierarchy_level,width} = 2
$comp_fields{EM,abbrev} = EM
$comp_fields{EM,group} =
$comp_fields{EM,name} = _2
$comp_fields{EM,title} = EM
$comp_fields{EM,type} = STRING
$comp_fields{EM,width} = 12
$comp_fields{FrameRelayUni,abbrev} = FrUni
$comp_fields{FrameRelayUni,group} =
$comp_fields{FrameRelayUni,name} = _279
$comp_fields{FrameRelayUni,title} = FrameRelayUni
$comp_fields{FrameRelayUni,type} = NUMERIC
$comp_fields{FrameRelayUni,width} = 5
$comp_fields{ifIndex,abbrev} = ifI
$comp_fields{ifIndex,group} = IfEntryProv
$comp_fields{ifIndex,name} = 3175
$comp_fields{ifIndex,title} = ifIndex
$comp_fields{ifIndex,type} = NUMERIC
$comp_fields{ifIndex,width} = 5
$comp_fields{ifAdminStatus,abbrev} = ifState
$comp_fields{ifAdminStatus,group} = IfEntryProv
$comp_fields{ifAdminStatus,name} = 3177
$comp_fields{ifAdminStatus,title} = ifAdminStatus
$comp_fields{ifAdminStatus,type} = STRING
$comp_fields{ifAdminStatus,width} = 7
$comp_fields{customerIdentifier,abbrev} = cid
$comp_fields{customerIdentifier,group} = \
       CustomerIdentifierData
$comp_fields{customerIdentifier,name} = 96
$comp_fields{customerIdentifier,title} = \
       customerIdentifier
$comp_fields{customerIdentifier,type} = STRING
$comp_fields{customerIdentifier,width} = 4
$comp_fields{numberOfEmissionQs,abbrev} = \
       numberOfEmissionQs
$comp_fields{numberOfEmissionQs,group} = \
       EmissionPriorityQs
$comp_fields{numberOfEmissionQs,name} = 9332
$comp_fields{numberOfEmissionQs,title} = \
```

```
            numberOfEmissionQs
    $comp_fields{numberOfEmissionQs,type} = STRING
    $comp_fields{numberOfEmissionQs,width} = 1
```

Sample Perl EPI script

This section provides two sample Perl EPI script: Passport card inventory (Command interface), and Passport DLCI configuration report (NRS interface).

Passport Card inventory sample

The following Perl script uses the command interface to produce a card inventory of a specified Passport node (using the current user session). This example is available in the Preside Multiservice Data Manager (MDM) load in `/opt/MagellanNMS/cfg/macros/nms/src/PPCardInv.pl`.

```
#!/opt/MagellanContrib/bin/perl

use lib “/opt/MagellanNMS/lib”;
use EPI qw(:all);

# Extract the Arguments.
if ( @ARGV < 2 ) {
  print “ppcarddrop <group> <passport>\n”;
  exit 1;
} else {
  $grp = $ARGV[0];
  $mod = $ARGV[1];
}

# initializes the Command interface
NMSCmdInit();
if ( NMSCmdConnect() != 1 ) {
  print “Failed to connect to the Command Session
servers.\n”;
```

```
  exit 1;
}

print "
                                Passport Card Inventory
-------------------------------------------------------
---------------------------
Node        Card Type        Inserted    Serial #
Firm. Rev.     LP
------------ ---- ------------ ------------
-------------- -------------- ------\n";

# do we have access to the node
NMSCmdSendCommand($grp, "$mod h -v(d) shelf card");
NMSCmdRecvFullReply();
if ( NMSCmdPatternMatch("Shelf Card") != 1 ) {
  print "\n*** Passport node $mod does not seem to be
reachable.\n";
  exit 1;
}
if ( NMSCmdPatternMatch("noTabular") ) {
  $opt = "-noTabular";
} else {
  $opt = "";
}

# get the number of slots from the shelf component
NMSCmdSendCommand($grp, "$mod d $opt shelf
numberOfSlots");
while (NMSCmdRecvNextLine()) {
  if ( NMSCmdGetColumn(1, "numberOfSlots") ) {
```

```
    $numberOfSlots = NMSCmdGetColumn("-out", 3);
  }
}

# get the needed attributes from all card components
$cardType = "";
NMSCmdSendCommand($grp,"$mod d $opt shelf card/*
cardType,insertedCardType,serialNumber,activeFirmware
Version,currentLP");
SLOT: while ( NMSCmdRecvNextLine() ) {
  $col = NMSCmdGetColumn("-out", 1);
  $val = NMSCmdGetColumn("-out", 3);
  SWITCH: {
    $col eq "cardType" \
      && do { $cardType              = $val };
    $col eq "insertedCardType" \
      && do { $insertedCardType      = $val };
    $col eq "activeFirmwareVersion" \
      && do { $activeFirmwareVersion = $val };
    $col eq "serialNumber" \
      && do { $serialNumber          = $val };
    $col eq "currentLP" \
      && do { $currentLP             = $val };
    $col eq "Shelf" \
      && do {
      # this is the name of a card (either the first or
      # another one in the list
      if (   ( length($cardType) > 0 )\
          && (   ( $cardType       ne "none" ) \
```

```
                || ( $insertCardType ne “none” ) ) ) {
            printf (“%-12s %-4s %-12s %-12s %-14s %-14s
%s\n”, \
                    $mod, $card, $cardType, \
                    $insertedCardType, $serialNumber, \
                    $activeFirmwareVersion, $currentLP);
            $cardType=””;
         }
        # extract the card number from the name and stop
if max
         NMSEPIPatternMatch(“Card/”, \
                        NMSCmdGetColumn(“-out”,2), ””);
         $card=”$NMSEPI_OUTPUT_MATCH”;
         last SLOT  if ( $card > $numberOfSlots );
       }
     }
}

# print the last card’s info if need be
if (   ( length $cardType  > 0 ) \
    && (   ( $cardType         ne “none” ) \
         || ( $insertedCardType ne “none” ) ) ) {
  printf (“%-12s %-4s %-12s %-12s %-14s %-14s %s\n”,
            $mod, $card, $cardType, $insertedCardType,
             $serialNumber, $activeFirmwareVersion,
             $currentLP);
}

exit 0;
```

Passport DLCI Configuration Report sample

The following Perl script uses the NRS interface to produce a produce a simple report all configured Frame Relay DLCIs in the network and their major quality-of-service parameters. This example is available in the MDM load `/opt/MagellanNMS/cfg/macros/nms/src/DlciScan.pl`.

```
#!/opt/MagellanContrib/bin/perl

# Load the NMS EPI extensions for reporting

use lib "/opt/MagellanNMS/lib";

use EPIR qw(:all);

use EPI  qw(:all);


# Initialize an NRS interface.

NMSNRSInit();


# Initiate a query for the various FrameRelay Service
# Parameter components.

NMSNRSStartQuery("ppc/FrameRelayUni", \
                 "ppc/EM-FrUni-Dlci-Sp");


# Report on all latest configurations (file system
# date) for all nodes.

NMSNRSAddSource(-latest=>"ppc", ".*");


# Scan the NRS database for all matching components

while ( NMSNRSFetchNextComponent(-var=>comp,
                                 -byname) ) {

     # Extract the major fields by name
     # (the associative array has them by type)

     if ( $NMSEPI_NRS_COMPABBREV eq "FrUni" ) {

          $cid = $comp{customerIdentifier};
```

```
            next;
        } else {
            $cir = $comp{committedInformationRate};
            $bc  = $comp{committedBurstSize};
            $be  = $comp{excessBurstSize};
        }
        # Print the component information in
        # comma-separated format for the target system
        NMSEPIConvertCompId(-disp, $NMSEPI_NRS_COMPID);
        print  "$NMSEPI_COMP_ID,$cid,$cir,$bc,$be\n";
    }
```

Chapter 5
C/C++ Embedded Programming Interface

This section describes the C/C++ version of the Embedded Programming Interface (EPI). This chapter contains the following:

Code conventions

There are two code conventions used in this chapter:

- \
  A back slash (\) indicates that the line of code is continued on the next line space.

- //
  A message line that starts with a double-slash is treated as a comment (C++ convention). For C code, comments are also indicated with a pair of /* */ delimiters.
- Except where indicated, all examples are specified using C (which makes them compatible with C++).

Integration methodology

The C/C++ version of the EPI interface is a public C version of the underlying interfaces used, value added, and specialized by the scripting language versions of EPI.

To minimize the number of Preside Multiservice Data Manager (MDM) internal classes that have to be published and to ensure the stability and independence of the interface, a C version of the interface is provided even though MDM internally uses C++ object-oriented code. For the same reason, no third party dependency is introduced by the C/C++ EPI other than the standard UNIX and C/C++ libraries available in Solaris. The various EPI C/C++ interfaces typically consist of a routine to construct the interface (this is equivalent to a C++ constructor or to the scripting language EPI **`Init`** call). This routine returns an opaque pointer, type **`EPIInterface`** (*`void*`*), which must be passed to each function of that interface (equivalent to the *`this`* pointer of C++ or the **`-iname`** option of the scripting language EPI calls). Finally, each interface provides a routine to free the interface (this is equivalent to a C++ destructor or the scripting language EPI Drop calls).

For your convenience, a simple C++ class mapping is also provided that mainly recasts the C EPI interfaces as inlined C++ class methods (to the point that both the C function calls and C++ methods are described at the same time in this chapter). This class mapping hides the opaque interface pointer. The provided class hierarchy is very flat, aligning to the major exported EPI interfaces.

To compile your programs with the C/C++ EPI, you need to include the MDM EPI C or C++ include file in your code modules. The EPI extensions are divided into two sets. For C code, you need the following line to access the base API, Command, Customer Database, and Miscelaneous interfaces:

```
#include <stdlib.h>
#include <EPIPublic.h>
```

To use the RTAC and NRS interfaces, you also need to include:

```
#include <EPIPublicR.h>
```

For C++, you can either use the same as above or use the following line if you also want to use the C++ class mappings for the base interfaces:

```
#include <stdlib.h>
#include <EPIPublic.hxx>
```

and the following one for the RTAC and NRS interfaces:

```
#include <EPIPublicR.hxx>
```

In both cases, you will also need to specify the following options to your compile lines to access these files and compile in compatibility mode:

```
<compile command> -I/opt/MagellanNMS/lib \
            -compat \
            <other options>
```

where `<compile command>` depends on your compiler installation.

You then need to link your program with the MDM EPI and the Xt and X11 libraries (the latter two are needed for the event loop meaning that the MDM code interacts as if you use it from X-Windows or CDE based code). For C++, the necessary command line is:

```
<link command> <other options> \
            -qoption ld -znodefs \
            -L/opt/MagellanNMS/lib \
            -R /opt/MagellanNMS/lib \
            -L/usr/openwin/include/X11 \
            -R /usr/openwin/include/X11 \
            -lEPIPublic -lXt -lX11
```

where `<link command>` depends on your compiler installation.

To use the RTAC and NRS interfaces, the necessary command line is:

```
<link command> <other options> \
            -qoption ld -znodefs \
            -L/opt/MagellanNMS/lib \
            -R /opt/MagellanNMS/lib \
            -L/usr/openwin/include/X11 \
            -R /usr/openwin/include/X11 \
            -lEPIPublicR -lXt -lX11
```

Note: The use of the Xt and X11 libraries does not force your applications to be graphical. They are needed primarily for the event dispatching mechanism they provide. This means that C/C++ EPI code will interwork with any Xt based systems in asynchronous mode. This also means that if your applications have their own non-Xt based event dispatching system, you will not be able to use C/C++ EPI in asynchronous mode.

If you are using C, you will also need to include the standard C++ Run Time library (libC.so) as it contains code needed by EPI:

```
<link command> <other options> \
            -znodefs \
            -L/opt/MagellanNMS/lib \
            -R /opt/MagellanNMS/lib \
            -lEPIPublic -lXt -lX11 \
            -L/usr/openwin/include/X11 \
            -R /usr/openwin/include/X11 \
            -L/usr/lib/ -R/usr/lib -lC
```

where `<link command>` depends on your compiler installation.

Here again, use the following command to use the RTAC and NRS interfaces:

```
<link command> <other options> \
            -znodefs \
            -L/opt/MagellanNMS/lib \
            -R /opt/MagellanNMS/lib \
            -lEPIPublicR -lXt -lX11 \
            -L/usr/openwin/include/X11 \
            -R /usr/openwin/include/X11 \
            -L/usr/lib/ -R/usr/lib -lC
```

These command line options are correct for Sun's SunPro C and C++ compilers. If you use a different compiler, you will surely have to adjust these command lines to match your setup.

Code using C/C++ EPI interfaces can only run on an MDM workstation (Solaris) and falls under the same release considerations in terms of needed software and patch levels.

WARNING
Thread safety
C/C++ EPI code is NOT thread safe. You therefore cannot use C/C++ EPI calls in multi-threaded code.

Interface

C/C++ EPI provides five groups of function calls:

- **Base**:
  Provides mapping to the EPI base routines and utilities as well as additional housekeeping routines.
- **Generic API Access**:
  Provides access to the Generic Application Programming Interface (API) commands and replies.
- **Specialized API Access**:
  Provides specialized and value-added access to specific API Providers such as the Alarm and Status, Network Model, and Host Group Directory Service (HGDS) APIs.
- **Command Access**:
  Provides access to the Preside Multiservice Data Manager (MDM) command macro capabilities.
- **Customer Database Access:**
  Provides access to the MDM Customer Database capabilities.
- **Real-Time Alarm Collection (RTAC)**
  Provides access to the spooled alarms collected by RTAC.

- **Network Reportinng System (NRS)**
  Provides access to the network-wide configutation data collected by NRS.

With the C++ class mappings, the last three also map to specific C++ classes; `NMSAPI` and its derivatives `NMSGMDRAPI`, `NMSNMAPI`, and `NMSHGDSAPI`, `NMSCmd`, and `NMSCdb`.

Command usage information

The following information applies to the provided functions:

- **command argument**
  You need to specify all function arguments in the indicated order. In the C++ mappings, some arguments specify default values.
- **return codes**
  C/C++ EPI provides common function return codes to indicate the successful or failed execution of the call. Functions usually return `EPI_SUCCESS` (0) upon success and a negative number upon failure. The possible values for the return codes follow:

`EPI_SUCCESS`	`(0)`	success
`EPI_FAILED`	`(-1)`	operation failed
`EPI_TIMEOUT`	`(-2)`	operation timed out
`EPI_NO_CONNECTION`	`(-3)`	interface not created, initialized or connected to server
`EPI_BUSY`	`(-4)`	a query is already active
`EPI_NO_QUERY`	`(-5)`	no active query
`EPI_END`	`(-6)`	end of query (no more replies)
`EPI_BADARG`	`(-7)`	pad parameter passed to function call

- **time-outs**
  Many functions allow you to specify a timeout value indicating the amount of time to wait for a reply before giving up. The following values are supported:

`EPI_TIMEOUT_FOREVER`	`(-1)`	the function is blocking and forever waits for a reply.

`EPI_TIMEOUT_POLL` `(0)`	the function does not wait and returns immediately if a reply is not already available.
`other numerical values`	the function waits at most for the specified number of seconds for a reply to become available.

- **returned strings and text**
  Some functions return text string values either directly or as part of structures. These returned values **must not be modified or deleted**. Their value may change or the actual pointer become invalid upon further EPI function calls or after returning to the event loop by returning from a callback function. You must make a copy of your own if you plan in keeping the value for a long time.

Differences with the scripting language EPIs

The C/C++ level interface is similar to the scripting language ones. There are nevertheless some differences between the scripting and the C/C++ EPI interfaces:

- different return code values.
  See “Command usage information” (page 354).
- no global/environment variables to return values from calls or callbacks. Values are returned directly by the function calls. In callbacks, one must extract the values using the provided function calls.
- no command help (`-h` option), output (`-out`), or interface name (`-iname`) options.
  These an simply inappropriate for a compiled language interface. Named interfaces are not needed as multiple individual interfaces can be directly created. In addition to this document, the actual include files; `EPIPublic.h` and `EPIPublic.hxx` contain documentation on the various supported calls.
- no standard error stream output of error messages.
  Errors are indicated through return codes only.
- no Long Arithmetic function calls.
  Long arithmetic (`long long`) can be performed directly in C/C++ on Solaris.

- no Timer Callback function calls.
  `XtAddTimer` (from the Xt library) can be invoked directly from C/C++.
- no associative array storage for return values.
  Functions are provided to retrieve all return values. They can be used to fill up your own data-structures.

Base

In general, the first task of a C/C++ EPI program is to initialize an interface. This function **must** be invoked before any other C/C++ function that uses Preside Multiservice Data Manager (MDM) Inter-Process communications (Context manipulation, API, Command, and Customer Database interfaces):

- **EPIResult NMSEPIInit**(*char* * aName); (C/C++)
  Initializes the EPI environment and provides the specified name, <aName>, as a process name.
- **EPIResult NMSEPITerm**(); (C/C++)
  Drops all current API and command interfaces and terminates the IPC environment. This is sometimes required when another EPI program (which makes and maintains its own API or command interfaces) is reused and invoked in the same UNIX process. The second program would otherwise fail because of duplicate server connections.

The following Preside Multiservice Data Manager (MDM) utilities are supported:

- *char* * compid = **NMSEPIConvertCompId**(
  char * compId,
  EPICompIdConversion conversion); (C/C++)
  Converts the specified component ID, and returns the string result (do not modify nor free) on success or *NULL* on failure. The conversion codes, <conversion>, are as follows:

EPI_CANON_CVT	(0) converts to canonical API format (for example, "PM AM1 PE 1 PI 1").
EPI_DISPLAY_CVT	(1) converts to display format (for example, "PM/AM1 PE/1 PI/1").
EPI_TYPE_CVT	(2) extracts the module/link type (for example, "PM" or "NL").

EPI_MNEMONIC_CVT (3) extracts the module mnemonic (for example, "AM1").

EPI_EP1_CVT and **EPI_EP2_CVT** (4) (5) extract the first and second link endpoints in canonical API format.

EPI_DPN_CVT (6) converts a DPN-100 OA or a PE/PI/PO component ID to a form suitable for commands (<mnemonic> [pe <pe#>|<pi #> [<po #>]]). (For example, "AM1 11" is the output for an input of "PM/AM1 PE/11 PI/11".)

EPI_SWITCH_CVT (7) returns the module-level component ID in canonical API format (for example, "PM AM1" for an input of "PM/AM1 PE/11").

EPI_OMNI_CVT (8) returns the Preside Multiservice Data Manager (MDM) HP-OpenView DeskTop compatible component ID (similar to display format except for link names).

Examples (C/C++)

```
char * ep1 = NMSEPIConvertCompId(linkId, EPI_EP1_CVT);
printf("Endpoint1: %s\n", ep1);

char * dpn = NMSEPIConvertCompId(portId, EPI_DPN_CVT);
sprintf(cmd, "%s enable", dpn);
NMSCmdSendCommand(interface, cmd);
```

- *int* compare = **NMSEPICompareCompIds**(
  char * comp1,
  char * comp2); (C/C++)

  Compares the two component IDs and returns 0 if they are identical, <0 if the first component ID precedes the second, and >0 if the second precedes the first. This is an intelligent comparison that is aware of numerical versus textual instance value sorting order differences.

- *char* * timstr = **NMSEPIConvertTime**(*char* * inTime,
  EPITimeConvertion cvt,
  int isSTC, *char* * param); (C/C++)

  Converts the specified time string, `inTime`, and returns the resultant string in a static buffer that must not be modified nor freed. The input time string can be in the following formats:

  API (`YYYY MM DD HH MM SS`)
  Common Alarm (`YY-MM-DD HH:MM:SS`)
  UNIX Epoch (`<number of seconds since 1970>`)
  Passport reply (`YYYY-MM-DD HH:MM_SS`)

  The returned time is in the same time frame as the input one except when `<isSTC>` is set to a nonzero value or when the `EPI_STC_TCVT` conversion is performed. If `<isSTC>` is set to a nonzero value, the input time is assumed to be in Standard Time Coordinates, that is, Greenwich Mean Time (GMT) If `<isSTC>` is set to a nonzero value and `EPI_STC_TCVT` is not used, then the output time is converted from STC to local workstation time. If the input time is `NULL`, the current workstation time is used and `<isSTC>` is ignored.

  The conversions are as follows:

`EPI_API_TCVT`	`(0)`	produces the time in API format
`EPI_STC_TCVT`	`(1)`	produces the time in API format converted to Standard Time Coordinates
`EPI_EPOCH_TCVT`	`(2)`	returns the time as a UNIX Epoch value (number of seconds since 1970)
`EPI_FTIME_TCVT`	`(3)`	returns the time in the format specified as the `<param>` argument (see `man -s3c strftime`, 100 characters maximum)

EPI_UNIX_TCVT	(4)	returns the time as the default time format for the workstation's LOCALE (see `man -s3c ctime`)
EPI_ALARM_TCVT	(5)	returns the time in Common Alarm format
EPI_PASSPORT_TCVT	(6)	returns the time in Passport reply format
EPI_DAY_OFFSET_TCVT	(7)	returns the time in API format after applying the positive or negative offset in days specified in the `<param>` string
EPI_SECONDS_OFFSET_TCVT	(8)	returns the time in API format after applying the specified positive or negative offset in seconds specified in the `<param>` string
EPI_MIDNIGHT_OFFSET_TCVT	(9)	returns the number of seconds from the previous midnight and the specified time (or current time)
EPI_STC_OFFSET_TCVT	(10)	returns the number of seconds between the local workstation time and the coordinated universal time (UTC) and, if needed, taking daylight savings time into consideration. The offset is positive going west from UTC (same as UNIX timezone/ altzone)

Example(C/C++)

```
char * tim = NMSEPIConvertTime(NULL,
                EPI_DAY_OFFSET_TCVT, 0, "7");
printf("%s\n", tim);
```

produces something like (the time 7 days ago):
`2000 03 24 17 01 19`

```
char * tim = NMSEPIConvertTime("2000 03 24 17 01 19",
              EPI_EPOCH_TCVT, 0, NULL);
printf("%s\n", tim);
```

produces something like (the same as a UNIX epoch):
`953935279`

```
char * tim = NMSEPIConvertTime("953935279",
              EPI_FTIME_TCVT, 0,
              "Time was: %a %b %d %l:%M%p");
printf("%s\n", tim);
```

produces something like (the same using a custom format):
`Time was: Fri Mar 24  5:01PM`

- *char* * string = **NMSEPIPatternMatch**(
  char * pattern,
  char * target,
  char * substitute,
  int all); (C/C++)

  Performs pattern matching of `<pattern>` against `<target>` and returns the string representing the matching part (do not modify nor free) on success or *NULL* on failure. The `<pattern>` is specified in grep syntax. You can include one `\\(` delimited sub-pattern (note the double backslash needed to work around C/C++'s string escaping mechanism). The text matching the sub-pattern is returned if one is specified instead of the entire match portion. If you specify a non-`NULL` `<substitute>`, the matching substring is replaced by it. If `<all>` is nonzero, the function substitutes all matching substrings.

  Example (C/C++)

```
/* extract the PE number */
char * string = NMSEPIPatternMatch( \
              "pe \\(\\[0-9\\]*\\)down",
              output, NULL, 0);
if (string)
   printf("PE# %s is down.\n", res);
...
```

- *char* * value = **NMSEPIGetContext**(
  EPIContextDomain domain,
  char * varname); (C/C++)
  EPIResult NMSEPISetContext(
  EPIContextDomain domain,
  char * varname,
  char * varvalue); (C/C++)
  Respectively gets or sets an Preside Multiservice Data Manager (MDM) context variable, `<varname>`, from or into the specified MDM Context Domain, `<domain>`. `EPI_GetContext` returns the variable value (do not modify nor free) on success or *NULL* on failure. `NMSEPISetContext` returns `EPI_SUCCESS` or `EPI_FAILED`. The supported Context Domains are:

`EPI_USER_CONTEXT`	`(0)`	the current User Session Context
`EPI_WS_CONTEXT`	`(1)`	the Workstation Wide Context
`EPI_SERVICE_CONTEXT`	`(2)`	the Service Selection Context

  Note: If you have not initialized an API, Command, or Customer Database interface prior to invoking one of these two commands then you must first invoke `NMSEPIInit` in order to properly initialize the Preside Multiservice Data Manager (MDM) communication environment.

  Example (C/C++)

```
char * ctx = NMSEPIGetContext(EPI_USER_CONTEXT, \
                              "DPN_QUICK_STEP");
if (ctx) printf("Hot context: %s\n", ctx);
```

- *void* **NMSEPIEventLoop**(); (C/C++)
  Initiates an Xt-based event loop for asynchronous message handling. This command never returns. The processing scripts are then performed by the callback functions that are bound to the API and other interfaces (see `NMSEPIRegisterContextInterest`, `NMSAPIBindCallback`, `NMSCmdBindCallback`, and `NMSCdbBindCallback`). This function is equivalent to invoking the `XtAppMainLoop` Xt function.

- **EPIResult NMSEPIRegisterContextInterest**(
  EPIContextDomain domain,
  EPIContextCallback ctxCb,
  char ** varnames); (C/C++)

  Binds the specified callback function <`ctxCb`> to the specified Preside Multiservice Data Manager (MDM) Context variables <`varnames`> (a *NULL* terminated array of strings) and <`domain`> (see `NMSEPIGetContext` above). This function will be executed whenever one of the named variable changes value and the program is in an event loop (`NMSEPIEventLoop`). The callback function has the following signature (C/C++):

  ```
  void (*EPIContextCallback)(int length, char * varname,
                             char * varvalue);
  ```

  where the arguments identify the variable name, value and value length.

  Note: Contrary to the other EPI C/C++ calls, **the <`varname`> and <`varvalue`> strings must be freed with *`free`*`()` if non-*`NULL`* before returning from the callback**.

  Only one such callback can be registered. To de-register the callback, call this function with *NULL* for the <`varnames`> array.

  Example (C)

  ```
  void
  componentHotContext(int l, char * vname, char * vval)
  int l;
  char * vname;
  char * vval;
  {
     /* react to the hot context selection
      whose component name value is in vval */
     ...
     /* free the values if non-NULL */
     if (vname) free(vname);
     if (vval) free(vval);
  }
  ...
  char * [] vnames = { "DPN_QUICK_STEP", NULL };
  NMSEPIRegisterContextInterest(EPI_USER_CONTEXT, \
  ```

```
                componentHotContext, vnames);
...
NMSEPIEventLoop();
```

Example (C++)

```
void
componentHotContext(int l;char * vname;char * vval)
{
    // react to the hot context selection
    // whose component name value is in vval
    ...
    // free the values if non-NULL
    if (vname) free(vname);
    if (vval) free(vval);
}
...
char * [] vnames = { "DPN_QUICK_STEP", NULL };
NMSEPIRegisterContextInterest(EPI_USER_CONTEXT, \
                componentHotContext, vnames);
...
NMSEPIEventLoop();
```

Generic API access

These function calls provide access to the Generic API at the same level as if you were using the Generic API Provider utility, *genapi*, directly. At this level, no special knowledge of the individual API types is required; all queries can be handled in a generic way. (For more information, see NTP 241-6001-200 *Preside MDM Application Programming Interface Primer*.

Generic API Access commands are used in the following sequence:

1 Initialize an API interface.

2 Connect to the API server.

3 If the server requires it, send a `REGISTER` message and wait for its reply.

4 Send an API query.

5 Receive the next reply and extract the needed information.

6 Disconnect from the API server.

7 Drop the API interface.

Multiple queries can be issued, but individual API interfaces will handle these queries synchronously, one at a time (except for sieve event notifications). It is possible, however, to create multiple API interfaces (each one resulting in its own connection to the server and therefore capable of independent, parallel queries). Note also that a single API interface can be reused (disconnected and reconnected to another server).

On the C++ side, the Generic API Interface is represented as a class, `NMSAPI`, that abstracts the interface opaque pointer and provides the same functionality as the C function calls but as inline methods. It is possible to derive from this class to add in your own capabilities and indeed C/C++ EPI does so itself to derive the specialized API interface described in "Specialized API access" (page 375).

Generic API Access commands

The following Generic API Access functions and methods are provided:

- **EPIInterface NMSAPIInit**(
  char * dictionary,
  int ssid); (C)
  NMSAPI::**NMSAPI**(
  char * dictionary,
  int ssid = 0); (C++ constructor)
  Initializes a Generic API interface using the indicated API dictionary (the value of <ssid> should always be `0`) and returns a pointer to it. The following constants are provided to name the most typical API dictionaries:

`EPIGenAPI_NM_DICT`	Network Model API
`EPIGenAPI_GMDR_DICT`	Alarm & Status API
`EPIGenAPI_HGDS_DICT`	Host Group Directory API
`EPIGenAPI_IMDR_DICT`	Inbound Alarm & Status API
`EPIGenAPI_NDAM_DICT`	Alarm & Status API, NDAM server

Note: The specialized API access interfaces provide their own version of this command, which hides the detail of their specific API dictionary path. For more information, see “Specialized API access” (page 375).

- *void* **NMSAPISetUserData**(
  EPIInterface api,
  void * userData); (C)
  void **NMSAPI**::**setUserData**(
  void * userData); (C++)

  void * **NMSAPIGetUserData**(
  EPIInterface api); (C)
  void * **NMSAPI**::**getUserData**(); (C++)
  void **NMSAPISetName**(
  EPIInterface api,
  char * name); (C)
  void **NMSAPI**::**setName**(
  char * name); (C++)

  char * **NMSAPIGetName**(
  EPIInterface api); (C)
  char * **NMSAPI**::**getName**(); (C++)
  These routines allow you to associate and extract arbitrary user data and a name to the API interface.

- *void* **NMSAPIDrop**(
  EPIInterface api); (C)
  virtual **NMSAPI**::~**NMSAPI**(); (C++ destructor)
  Destroys the specified API interface to clear up its connection and reclaim its allocated memory.

  Note: The C++ specialized API access interfaces provide their own version of the virtual class destructor.

- **EPIResult NMSAPIConnect**(
  EPIInterface api,
  char * servName,
  char * hostName); (C)
  EPIResult NMSAPI::**connect**(
  char * servName,

char * hostName = “localhost”); (C++)

Connects the interface to the named API server, <serverName>, and specified Preside Multiservice Data Manager (MDM) host, <serverHost> (if *NULL* is specified -- or the argument is missing in C++ -- <serverHost> defaults to “localhost”).

The following service name constant are defined for your convenience:

EPIGenAPI_NM_SERVICE	Network Model API Server
EPIGenAPI_GMDR_SERVICE	Alarm&Status API GMDR Serve
EPIGenAPI_HGDS_SERVICE	Host Group Directory Service API Server
EPIGenAPI_IMDR_SERVICE	Inbound Alarm&Status API IMDR Server
EPIGenAPI_NDAM_SERVICE	Alarm&Status and Network Model API from Network Data Access Manager Server

The following constants can be used to construct alternate GMDR and IMDR service names if appropriate to your Preside Multiservice Data Manager (MDM) configuration;

EPIGenAPI_GMDR_PREFIX set to “GMDR_”

EPIGenAPI_IMDR_PREFIX set to “IMDR_”

EPIGenAPI_NMDR_PREFIX set to “NMDR_”

The following host-name constants are also defined to automatically use the corresponding Service Selected host for the corresponding selection area:

EPI_GMDR_SELECTED_HOST	Surveillance Service Selected host
EPI_NM_SELECTED_HOST	Network Model Service Selected host.
EPI_EM_SELECTED_HOST	Passport Network Access Service Selected host
EPI_DPN_SELECTED_HOST	DPN Network Access Service Selected host.
EPI_DPNARCH_SELECTED_HOST	DPN Configuration Management Service Selected host.

These last constants can also be used with the `NMSEPISetContext`, `NMSEPIGetContext`, and `NMSEPIRegisterContext` functions to manipulate the corresponding Service Selection context variables.

Example (C)

```
EPIInterface almapi = NMSAPIInit(
            EPIGenAPI_GMDR_DICT, 0);
if( almapi ) {
   if (NMSAPIConnect(almapi, "GMDR", "localhost")
            == EPI_SUCCESS) {
   ... /* connection successful */
```

Example (C++)

```
NMSAPI * almapi = new NMSAPI(
            EPIGenAPI_GMDR_DICT, 0);
if( almapi ) {
   if (almapi->connect(almapi, "GMDR")
            == EPI_SUCCESS) {
   ... // connection successful
```

- **EPIResult NMSAPIDisconnect**(
  EPIInterface api); (C)
  EPIResult NMSAPI::**disconnect**(); (C++)
  Disconnects the interface from its current API server.

- **EPIResult NMSAPIIsOK**(
  EPIInterface api); (C)
  EPIResult NMSAPI::**isOK**(); (C++)
  Returns current connection status of the interface (`EPI_SUCCESS`, `EPI_FAILED`, or `EPI_NO_CONNECTION`).

- **EPIResult NMSAPIRegister**(
  EPIInterface api,
  char * useId,
  char * passwd,
  char * attrs); (C)
  EPIResult NMSAPI::**registerAPI**(
  char * userId,
  char * passwd = NULL,
  char * attrs = NULL); (C++)
  Sends a `REGISTER` to the interface with the specified parameters. If attrs is non-`NULL`, it provides a string containing additional attribute specifications in API syntax, one per line (for example, the "_attr: userCapability E mdInject" line needed to register to the IMDR server with alarm injection capabilities).

  This comand is actually a full sequence where the `REGISTER` command is sent and its reply waited for. Consequently, the return code reflects on the registration's success, not only in sending the command.

  Example (C)

```
...
   if (   (NMSAPIConnect(almapi, "IMDR", "localhost")
            == EPI_SUCCESS)
       && (NMSAPIRegister(almapi, "ticketIf", NULL,
            NULL) == EPI_SUCCESS) ){
   ... /* registration successful, ready to send
           commands */
```

  Example (C++)

```
...
   if (   (almapi->connect(almapi, "IMDR")
            == EPI_SUCCESS)
       && (almapi->registerAPI("ticketIf")
```

```
            == EPI_SUCCESS) ){
    ... // registration successful, ready to send
        // commmands
```

- **EPIResult NMSAPISendAPICommand**(
  EPIInterface api,
  char * command); (C)
  EPIResult NMSAPI::**sendAPICommand**(
  char * command); (C++)
  Sends a query to the interface. The query is specified as an ASCII string in API syntax.

  Example (C)

```
if ( NMSAPISendAPICommand(api, "_cmd: get\n\
_obj_type: network\n\
_obj_id: networkID S compRoot\n\
_scope: all\n\
_attr_id: all\n\
_filter:compID LEFT NI PM") == EPI_SUCCESS ) {
...
```

  Example (C++)

```
if ( api->sendAPICommand(api, "_cmd: get\n\
_obj_type: network\n\
_obj_id: networkID S compRoot\n\
_scope: all\n\
_attr_id: all\n\
_filter:compID LEFT NI PM") == EPI_SUCCESS ) {
...
```

- **EPIResult NMSAPIRecvAPIReply**(
  EPIInterface api,
  EPITimeout timeout); (C)
  EPIResult NMSAPI::**recvAPIReply**(
  EPITimeout timeout
  = **EPI_TIMEOUT_FOREVER**); (C++)
  Waits for and receives the next reply record from the server. A timeout can be specified (by default, it waits forever -- `EPI_TIMEOUT_FOREVER` or `-1`). With a timeout of `EPI_TIMEOUT_POLL`, or `0`, the command acts as a no-wait poll. Other values wait for the specified number of seconds.

- **EPIResult NMSAPISkipRestOfReply**(
  EPIInterface api,
  int * error); (C)
  EPIResult NMSAPI::**skipRestOfReply**(
  int * error = *NULL*); (C++)
  This function waits for and ignores all further replies until the end of response. If it finds an error record in the replies the error argument is set to `1` (`0` otherwise).

- **EPIGenAPI_RECORD_TYPES**
  NMSAPIGetAPIRecord(
  EPIInterface api); (C)
  EPIGenAPI_RECORD_TYPES
  NMSAPI::**getAPIRecord**(); (C++)
  Extracts the last received record's (`NMSAPIRecvAPIReply or NMSAPI::recvAPIReply`) Record Type enumeration token, one of:

`NONE`	Unknown record type
`REGISTER`	REGISTER message reply
`RESPONSE`	Response to a query
`EVENT`	Event notification from a sieve
`ENDRESP`	End of query or sieve creation
`ERROR`	Error reply
`END`	End of communication (dropping of API server connection)

  This function also resets the API field list to the beginning for `NMSAPIGetNextField/NMSAPI::getNextField`, `NMSAPIFindNextField/NMSAPI::findNextField,` and `NMSAPIFindNextAttr/NMSAPI::findNextAttr` (it can therefore be called several times to repeatedly scan the field/attribute list).

Example (C)

```
while ( NMSAPIRecvAPIReply(api,
                EPI_TIMEOUT_FOREVER) == EPI_SUCCESS) {
    EPIGenAPI_RECORD_TYPES rtype =
                NMSAPIGetAPIRecord(api);
    if ( rtype = ENDRESP ) {
                ... /* got end of response */
```

Example (C++)

```
while ( api->recvAPIReply() == EPI_SUCCESS) {
    EPIGenAPI_RECORD_TYPES rtype =
                api->etAPIRecord();
    if ( rtype = ENDRESP ) {
                ... // got end of response
```

- *char* * name = **NMSAPIRecordTypeName**(
  EPIGenAPI_RECORD_TYPES rtype); (C/C++)
  This utility function simply provides a token to text mapping for the API Record Type enumeration values. The mapping used simply maps each Record Type enumeration token to its textual name.

- **EPIGenAPIField** * **NMSAPIGetNextAPIField**(
  EPIInterface api); (C)
  EPIGenAPIField * **NMSAPI**::**getNextAPIField**(); (C++)
  Extracts the next API field from the last received reply. The returned structure pointer comtains the following members:

  EPIGenAPI_FIELD_TYPES ftype
  is the API field type whose enumeration token values are; **_ATTR**, **_END_RESP**, **_ERROR**, **_END**, **_OBJ_TYPE**, **_OBJ_ID**, **_USER_ID**, **_CAPABILITY**, **_INV_ID**, **_EVENT_TYPE**, **_TIME**, and **_SIEVE_ID**.

  *char * **name***
  is the API field name element (for **_ATTR** and **_OBJ_ID** fields only).

  *char * **type***
  is the API field type element (for **_ATTR** and **_OBJ_ID** fields only).

char * **value**
is the API field value.

Multiple-line attribute block values are returned as a single multiple-line value. Inapplicable fields may have *NULL* values. Do not modify nor free the returned text values.

Example (C)

```
NMSAPIGetAPIRecord(api);
...
EPIGenAPIField * fld;
while ( ( fld = NMSAPIGetNextAPIField(api) )
                != NULL ) {
   if (   (fld->ftype == _ATTR)
       && (strcasecmp(fld->name, "time") == 0) ) {
      printf("Time is: %s\n", fld->value);
...
```

Example (C++)

```
api->getAPIRecord();
...
EPIGenAPIField * fld;
while ( ( fld = api->getNextAPIField() )
                != NULL ) {
   if (   (fld->ftype == _ATTR)
       && (strcasecmp(fld->name, "time") == 0) ) {
      cout << "Time is: " << fld->value << endl;
...
```

- *char* * **NMSAPIFieldTypeName**(
  EPIGenAPI_FIELD_TYPES ftype); (C/C++)
  This utility function simply provides a token to text mapping for the API Field Type enumeration values. The mapping used simply maps each Field Type enumeration token to its textual name in lower-case characters and followed by a colon (for example, _ATTR maps to "_attr:").

- **EPIGenAPIField** * fld = **NMSAPIFindNextAPIField**(
  EPIInterface api,
  EPIGenAPI_FIELD_TYPES ftype); (C)
  EPIGenAPIField * fld = **NMSAPI**::**findNextAPIField**(

EPIGenAPI_FIELD_TYPES ftype); (C++)

Locates and returns the next API field of the type specified by `<ftype>` from the API field list of the last received API reply. The field is returned as described in for the `NMSAPIGetNextAPIField/ NMSAPI::getNextAPIField` call. The previous examples can be rewritten as follows:

Example (C)

```
NMSAPIGetAPIRecord(api);
...
EPIGenAPIField * fld;
while ( ( fld = NMSAPIFindNextAPIField(api,
              _ATTR) ) != NULL ) {
   if ( strcasecmp(fld->name, "time") == 0 ) {
      printf("Time is: %s\n", fld->value);
...
```

Example (C++)

```
api->getAPIRecord();
...
EPIGenAPIField * fld;
while ( ( fld = api->findNextAPIField(_ATTR) )
              != NULL ) {
   if ( strcasecmp(fld->name, "time") == 0 ) {
      cout << "Time is: " << fld->value << endl;
...
```

- **EPIGenAPIField** * fld = **NMSAPIFindNextAPIAttr**(
  EPIInterface api,
  char * aname); (C)

  EPIGenAPIField * fld = **NMSAPI**::**findNextAPIAttr**(
  char * aname); (C++)

  Locates and returns the next named attribute from the API field list of the last received API reply. The field is returned as described in for the `NMSAPIGetNextAPIField/NMSAPI::getNextAPIField` call. The previous example can be rewritten as follows:

  Example (C)

  ```
  NMSAPIGetAPIRecord(api);
  ...
  EPIGenAPIField * fld;
  while ( ( fld = NMSAPIFindNextAPIField(api,
  ```

```
                    "time") ) != NULL ) {
    printf("Time is: %s\n", fld->value);
...
```

Example (C++)

```
api->getAPIRecord();
...
EPIGenAPIField * fld;
while ( ( fld = api->findNextAPIField("time") )
                    != NULL ) {
    cout << "Time is: " << fld->value << endl;
...
```

- *void* **NMSAPIBindCallback**(
  EPIInterface api,
  EPICallBackFunction aCbFn); (C)
  void **NMSAPI**::**bindCallback**(
  EPICallBackFunction aCbFn); (C++)
  Binds the specified callback function to the API interface. This callback will be invoked whenever a new message is received from the server for this interface and the program is in an event loop (`NMSEPIEventLoop`). The callback mechanism acts as though the indicated function was called immediately after a call to `NMSAPIGetAPIRecord/ NMSAPI::getAPIRecord` The callback signature is:

  `typedef` *void (*`EPICallBackFunction`) (***EPIInterface** api, *char* * not_used, *int* rtype, *int* status*)*

  where `<api>` is an opaque pointer to the calling API interface (as returned by `NMSAPIInit/NMSAPI::NMSAPI`), `<rtype>` is the type of the record just received (compatible with the `EPIGenAPI_RECORD_TYPES` enumeration) and `<status>` is one of `EPI_SUCCESS`, for a successful reception, `EPI_NO_CONNECTION`, if the connection to the server was lost, and `EPI_FAILED`, for other error cases.

  In addition, the post-`Recv` functions (i.e., `NMSAPIGetAPIRecord/ NMSAPI::getAPIRecord`, `NMSAPIGetNextAPIField/ NMSAPI::getNextAPIField`, ...) can be used in the context of this callback to extract API fields and attributes from the received reply. The event loop is launched with `NMSEPIEventLoop` and never returns.

Example (C)

```
void
mycb{api, not_used, rtype_i, status}
void * api;
char *  no_used;
int     rtype_i;
int     status;
{
    EPIGenAPI_RECORD_TYPES rtype =
        (EPIGenAPI_RECORD_TYPES)rtype_i;
    /*...Process the reply...*/
}
/*... launch commands and create sieves ...*/
NMSAPIBindCallback(mycb);
...
NMSEPIEventLoop(); /* never returns */
```

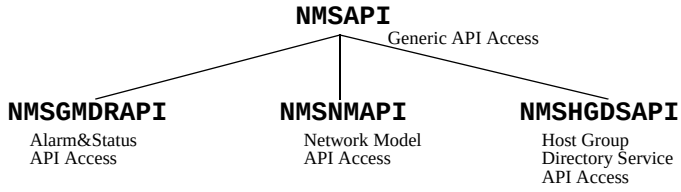
Example (C++)

```
void
mycb{void * apiP, char * not_used, int rtype_i,
        int status}
{
    NMSAPI  * api = (NMSAPI*)apiP;
    EPIGenAPI_RECORD_TYPES rtype =
        (EPIGenAPI_RECORD_TYPES)rtype_i;
    //...Process the reply...
}
//... launch commands and create sieves ...
api->bindCallback(mycb);
...
NMSEPIEventLoop(); // never returns
```

Specialized API access

This section describes additional commands as well as value-added versions of the Generic API Access commands that are built to interact with specific APIs and their features.

On the C++ side, the specialized API interfaces take the form of additional classes derived from the `NMSAPI` class. As the derivation is public, all `NMSAPI` methods are available to the derived classes and their instances. The result is a simple class hierarchy:



Alarm and Status API Access

This area adds commands to interact with the Alarm and Status API. In addition to a number of combination calls (for example, to create an alarm sieve or to inject alarms), it also supports automatic extraction of the major alarm attributes to support most forms of network automation.

The C++ derived class for Alarm&Status API access is `NMSGMDRAPI`.

Alarm and Status API Access commands

The following Alarm and Status API Access functions and methods are provided:

- **EPIInterface** api = **NMSGMDRAPIInit**(); (C)
  NMSGMDRAPI::**NMSGMDRAPI**(); (C++ constructor)
  This simplified form of the `NMSAPIInit/NMSAPI::NMSAPI` function/ constructor initializes an API interface to the Alarm and Status API server (GMDR).

- **EPIResult NMSGMDRAPIConnect**(
  char * host); (C)
  EPIResult NMSGMDRAPI::**connectOn**(
  char * host = “localhost”); (C++)
  Connects to the GMDR server on the specified location (`<host>`), or the local host (by default for C++). Use a value of `EPI_GMDR_SELECTED_HOST` to connect to the current Surveillance Service Selection.

Example (C)

```
EPIInterface api = NMSGMDRAPIInit();
...
/* connect to the service selected GMDR server */
if ( NMSGMDRAPIConnect(api, EPI_GMDR_SELECTED_HOST)
              == EPI_SUCCESS ) {
   ...
```

Example (C++)

```
NMSGMDRAPI * api = new NMSGMDRAPI();
...
// connect to the service selected GMDR server
if ( api->connect(EPIGenAPI_GMDR_SERVICE,
              EPI_GMDR_SELECTED_HOST)
              == EPI_SUCCESS ) {
   ...
```

- **EPIResult NMSGMDRAPICreateAlarmSieve**(
  EPIInterface api,
  int allflag,
  char * attrs,
  int sieveid); (C)

  EPIResult NMSGMDRAPI::**createAlarmSieve**(
  int allflag = 0,
  char * attrs = *NULL*,
  int *sieveid = *NULL*); (C++)

  This combination call creates a simplified alarm sieve and waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the major alarm attributes (`compId`, `time`, `severity`, `event`, `faultCode`, and `operatorData`) of all of the received alarms. However, if `<allflag>` is set to `1`, all of the attributes are extracted. You can add event filter (“`_attr: eventFilter SS <attribute> <operator> <type> <value>`”) and specific attribute extraction (“`_attr: eventInfo S`

<attribute>”) parameters in API syntax through the <attrs> argument (with a carriage-return character -- \n -- between each line). Use NULL is none is needed.

Each Alarm and Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned into the integer variable pointed to by the <sieveid> argument if non-*NULL*.

Example (C)

```
/* create an alarm sieve for all default attributes
 * plus relatedComponents but only SET alarms being
 * accepted */
if ( NMSGMDRAPICreateAlarmSieve(api, 0,
    “_attr: eventFilter SS event EQ E SET\n”
    “_attr: eventInfo S relatedComponents\n” ) {
    ... /* sieve creation successful */
```

Example (C++)

```
// create an alarm sieve for all default attributes
// plus relatedComponents but only SET alarms being
// accepted
if ( api->createAlarmSieve(0,
    “_attr: eventFilter SS event EQ E SET\n”
    “_attr: eventInfo S relatedComponents\n” ) {
    ... // sieve creation successful
```

The notifications from the sieve(s) can be collected with NMSAPIRecvAPIReply/NMSAPI::recvAPIReply or with NMSGMDRAPIRecvAlarm/NMSGMDRAPI::recvAlarm (with the EVENT record type).

Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **EPIResult NMSGMDRAPIRecvAlarm**(
  EPIInterface api,
  EPIGMDRAPI_Alarm ** alarm,
  EPITimeout timeout); (C)

  EPIResult NMSGMDRAPI::**recvAlarm**(

EPIGMDRAPI_Alarm ** alarm,
EPITimeout timeout
= **EPI_TIMEOUT_FOREVER**); (C++)

Enhances `NMSAPIRecvAPIReply/NMSAPI::recvAPIReply` by waiting for and receiving the next reply and automatically extracting the major alarm fields. The extracted fields are returned into a static area and a pointer to that area is returned as the `<alarm>` argument. This static area of type `EPIGMDRAPI_Alarm` contains the following fields whose values must not be modified nor freed (they are valid until the next API related function call):

char * **compId**;
is the alarm's component ID..

char * **time**;
is the alarm's time stamp in API format.

char * **severity**;
is the alarm's severity value.

char * **event**;
is the alarm's event type.

char * **faultCode**;
is the alarm's fault code.

char * **operatorData**;
is the alarm's operator data.

char * **rawState**;
is the alarm's raw state.

int **sieveId**;
is the sieve ID if the alarm is an event from a sieve (`0` otherwise).

Note: Some members may be *NULL* if the corresponding field is not present. As well, operatorData can consist of more than one line.

For information on the legal values of these fields, see NTP 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The other attributes, if any were specified, can be extracted with the NMSPAIGetAPIRecord/NMSAPI::getAPIRecord, NMSAPIGetNextField/NMSAPI::getNextField, NMSAPIFindNextField/NMSAPI::findNextField, and NMSAPIFindNextAttr/NMSAPI::findNextField commands.

A timeout can be specified. With a timeout of EPI_TIMEOUT_POLL (0), the command acts as a no-wait poll. A value of EPI_TIMEOUT_FOREVER (-1) waits forever. Other values wait for the specified number of seconds.

Example (C)

```
NMSAPICreateAlarmSieve(api, 0,
   "_attr: eventFilter SS event EQ E SET");
...
EPIGMDRAPI_Alarm * alarm;
if ( NMSAPIRecvAlarm(api, alarm,
              EPI_TIMEOUT_FOREVER) == EPI_SUCCESS ) {
   if (NMSAPIGetAPIRecord(api) == EVENT) {
      if (strcasecmp(alarm->severity,
              "critical") == 0) {
              ... /* handle the critical alarm */
```

Example (C++)

```
api->createAlarmSieve(0,
   "_attr: eventFilter SS event EQ E SET");
...
EPIGMDRAPI_Alarm * alarm;
if ( api->recvAlarm(alarm) == EPI_SUCCESS ) {
   if (api->getAPIRecord() == EVENT) {
      if (strcasecmp(alarm->severity,
              "critical") == 0) {
              ... // handle the critical alarm
```

- **EPIGMDRAPI_Alarm * NMSGMDRAPIExtractAlarm**(
  EPIInterface api); (C)
  EPIGMDRAPI_Alarm * **NMSGMDRAPI**::**extractAlarm**(); (C++)
  Extracts the Alarm information similarly to NMSGMDRAPIRecvAlarm/

`NMSGMDRAPI::recvAlarm` from the last received reply be it from a call to `NMSAPIRecvAPIReply/NMSAPI::recvAPIReply` or in a bound API Callback. The extracted fields, which must not be modified nor freed, are returned as a pointer to a static area valid until the next API related function call.

- *char* * disp = **NMSGMDRAPIFormatAlarm**(
  EPIInterface api,
  EPIGenAPI_ALARM_FORMATS format); (C)
  char * disp = **NMSGMDRAPI**::**formatAlarm**(
  EPIGenAPI_ALARM_FORMATS format
  = **EPI_ALARM_FULL_FORMAT**); (C++)
  This command produces the last received alarm (from `NMSAPIRecvAPIReply/NMSAPI::recvAPIRedply`, `NMSGMDRAPIRecvAlarm/NMSGMDRAPI::recvAlarm` or from a callback) in the Preside Multiservice Data Manager (MDM) Common Alarm Format (as used in the Alarm Display and Component Information Viewer tools). The alarm can be formatted, `<format>`, in either

`EPI_ALARM_TERSE_FORMAT`	one line summary
`EPI_ALARM_NORMAL_FORMAT`	includes Comment Data
`EPI_ALARM_FULL_FORMAT`	all information is displayed

  The default is `EPI_ALARM_FULL_FORMAT`. The returned text buffer contains the formatted alarm. Unlike other EPI strings, you need to ensure that your program releases this buffer when you are done with it.

- **EPIResult NMSGMDRAPIInjectAlarm**(
  EPIInterface api,
  char * compId,
  char * event,
  char * severity, *char* * faultCode,
  int notificationId,
  char * comment,
  char * time,
  char * attrs); (C)

EPIResult NMSGMDRAPI::**injectAlarm**(
char * compId,
char * event,
char * severity,
char * faultCode,
int notificationId,
char * comment,
char * time = NULL,
char * attrs = NULL); (C++)

Sends an alarm injection command with the following specified parameters:

`compId`
is the component ID.

`event`
is the alarm event (`set|clear|message`).

`severity`
is the severity of the alarm (`warning|minor|major|critical|cleared|indeterminate`).

`faultCode`
is the fault code in `AAAACCCC` format.

`notificationId`
is the sequence number (specified as an integer). If 0, a unique value will be provided.

`comment`
is the comment data string.

`time`
is the alarm time in API `"YYYY MM DD hh mm ss"` format, specify as *`NULL`* to let IMDR assign the current time.

attrs
can be used to assign additional attributes specified in API "`_attr: <name> <type> <value>`" line format (with `\n` as line separators).

Unspecified attributes take default values (see NTP 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*).

Note: This is not a combination call since the command does not wait for the server's reply, which must then be explicitly received or ignored.

Example (C)

```
NMSGMDRAPIInjectAlarm(api, "MYRTR WEST3", "major",
                "set", "A0000001", 23,
                "The router does not reply.", NULL,
                "_block: _attr operatorData S
Try: 12, Timeout: 5, IP: 33.24.1.1
_end_block:");
NMSAPISkipResOfReply(api, NULL);
```

Example (C++)

```
api->injectAlarm("MYRTR WEST3", "major",
                "set", "A0000001", 23,
                "The router does not reply.",
                "_block: _attr operatorData S
Try: 12, Timeout: 5, IP: 33.24.1.1
_end_block:");
api->skipRestOfReply();
```

- **EPIResult NMSGMDRAPICreateRawStateSieve**(
  EPIInterface api,
  char * attrs, *int* sieveid); (C)
  EPIResult NMSGMDRAPI::**createRawStateSieve**(
  char * attrs = *NULL*,
  int *sieveid = *NULL*); (C++)
  This combination call creates a simplified raw state change sieve and waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the `compId`, `time`, and `rawState` attributes. It is possible to add an event

filter (“_attr: eventFilter SS <attribute> <operator> <type> <value>”) with the <attrs> argument (with a carriage-return character -- \n -- between each line). Use NULL is none is needed.

Each Alarm and Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned into the integer variable pointed to by the <sieveid> argument if non-*NULL*.

Example (C)

```
int sieveId;
if ( NMSGMDRAPICreateRawStateSieve(api,
          “_attr: eventFilter SS rawState EQ E OOS”,
          &sieveId) == EPI_SUCCESS ) {
   ... /* sieve creation successful */
```

Example (C++)

```
int sieveId;
if ( api->createRawStateSieve(
          “_attr: eventFilter SS rawState EQ E OOS”,
          &sieveId) == EPI_SUCCESS ) {
   ... // sieve creation successful
```

The notifications from the sieve(s) can be collected with NMSAPIRecvAPIReply/NMSAPI::recvAPIReply or with NMSGMDRAPIRecvRawState/NMSGMDRAPI::recvRawState.

Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **EPIResult NMSGMDRAPIRecvRawState**(
  EPIInterface api,
  EPIGMDRAPI_RawState * rawstate,
  EPITimeout timeout); (C)
  EPIResult NMSGMDRAPI::**recvRawState**(
  EPIGMDRAPI_RawState * rawstate,
  EPITimeout timeout
  = **EPI_TIMEOUT_FOREVER**); (C++)
  Enhances NMSAPIRecvReply by waiting for and receiving the next

reply and automatically extracting the raw state change fields. The extracted fields are returned into a static area and a pointer to that area is returned as the `<rawstate>` argument. This static area of type `EPIGMDRAPI_RawState` contains the following fields whose values must not be modified nor freed (they are valid until the next API related function call):

`char * **compId**;`
the target component ID.

`char` `* **time**;`
is the notification time.

`char * **rawState**;`
the raw state value.

`int` `**sieveId**;`
is sieve ID if this is an event from a sieve (`0` otherwise).

For information on the legal values of these fields, see NTP 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*.

A timeout can be specified. With a timeout of `EPI_TIMEOUT_POLL` (`0`), the command acts as a no-wait poll. A value of `EPI_TIMEOUT_FOREVER` (`-1`) waits forever. Other values wait for the specified number of seconds.

Example (C)

```
NMSGMDRAPICreateRawStateSieve(api, NULL);
...
EPIGMDRAPI_RawState * rstate;
if ( NMSGMDRAPIRecvRawState(api, rstate,
           EPI_TIMEOUT_FOREVER) == EPI_SUCCESS ) {
   if (NMSAPIGetAPIRecord(api) == EVENT) {
     if ( strcasecmp(rstate->rawState, "OOS") == 0 ) {
           ... // handle the out-of-service component
```

Example (C++)

```
api->createRawStateSieve(NULL);
...
EPIGMDRAPI_RawState * rstate;
if ( api->recvRawState(rstate) == EPI_SUCCESS ) {
```

```
if (api->getAPIRecord() == EVENT) {
  if ( strcasecmp(rstate->rawState, "OOS") == 0 ) {
        ... // handle the out-of-service component
```

- **EPIGMDRAPI_RawState** * **NMSGMDRAPIExtractRawState**(
  EPIInterface api); (C)
  EPIGMDRAPI_RawState * **NMSGMDRAPI**::**extractRawState**();
  (C++)
  Extracts the Raw State information similarly to `NMSGMDRAPIRecvRawState/NMSGMDRAPI::recvRawState` from the last received reply be it from a call to `NMSAPIRecvAPIReply/NMSAPI::recvAPIReply` or in a bound API Callback. The extracted fields, which must not be modified nor freed, are returned as a pointer to a static area valid until the next API related function call.

Network Model API access

The Network Model (NM) API access allows you to initialize an API interface to the server and connect the interface to the host.

The C++ derived class for Netowrk Model API access is `NMSNMAPI`.

Network Model API Access commands

Network Model (NM) API Access provides only one additional function/ method:

- **EPIInterface** api = **NMSNMAPIInit**(); (C)
  NMSNMAPI::**NMSNMAPI**(); (C++ constructor)
  This simplified form of the `NMSAPIInit/NMSAPI::NMSAPI` function/ constructor initializes a Network Model API interface.

- **EPIResult NMSNMAPIConnect**(
  char * host); (C)
  EPIResult NMSNMAPI::**connectOn**(
  char * host = "localhost"); (C++)
  Connects to the Network Model server on the specified location (`<host>`), or the local host (by default for C++). Use a value of `EPI_NM_SELECTED_HOST` to connect to the current Network Model Service Selection.

Example (C)

```
EPIInterface api = NMSNMAPIInit();
...
/* connect to the local Network Model server */
if ( NMSNMConnect(api, "localhost")
              == EPI_SUCCESS ) {
   ...
```

Example (C++)

```
NMSNMAPI * api = new NMSNMAPI();
...
// connect to the local Network Model server
if ( api->connectOn("localhost") == EPI_SUCCESS ) {
   ...
```

Various Generic API Access commands can be used to send queries and receive their replies.

Host Group Directory Service API access

The Host Group Directory Service (HGDS) API Access allows you to extract data on the Passport Group configuration of the Preside Multiservice Data Manager (MDM). Some value-added and combination calls are added to the Generic API Provider along with automatic extraction of the Passport Member information.

The C++ derived class for Host Group Directory Service API access is `NMSHGDSAPI`.

HGDS API Access commands

The following HGDS API Access commands are provided:

- **EPIInterface** api = **NMSHGDSAPIInit**(); (C)
  NMSHGDSAPI::**NMSHGDSAPI**(); (C++ constructor)
  This simplified form of the `NMSAPIInit/NMSAPI::NMSAPI` function/constructor initializes a Host Group Directory Service API interface.

- **EPIResult NMSHGDSAPIConnect**(
  char * host); (C)
  EPIResult NMSHGDSAPI::**connectOn**(
  char * host = "localhost"); (C++)
  Connects to the Host Group Directory server on the specified location (`<host>`), or the local host (by default for C++). Use a value of

`EPI_EM_SELECTED_HOST` to connect to the current Passport Access Service Selection or `EPI_DPN_SELECTED_HOST` to connect to the current DPN Access Service Selection.

Example (C)

```
EPIInterface api = NMSHGDSAPIInit();
...
/* connect to the service selected HGDS server */
if ( NMSHGDSAPIConnect(api, EPI_EM_SELECTED_HOST)
              == EPI_SUCCESS ) {
   ...
```

Example (C++)

```
NMSHGDSAPI * api = new NMSHGDSAPI();
...
// connect to the service selected HGDS server
if ( api->connect(EPI_EM_SELECTED_HOST)
              == EPI_SUCCESS ) {
   ...
```

- **EPIResult NMSHGDSAPISendQuery**(
  EPIInterface api,
  EPIHGDSScope scope,
  char * parameter); (C)
  EPIResult NMSHGDSAPI::**sendQuery**(
  EPIHGDSScope scope,
  char * parameter = *NULL*); (C++)
  Sends an HGDS API query as specified by the `<scope>` argument and optional `<parameter>`. The supported scopes are:

`EPI_HGDS_GET_PARENT`	extracts the groups for the identified host name
`EPI_HGDS_GET_CHILDREN`	extracts the member hosts for the identified group
`EPI_HGDS_GET_GROUP`	extracts the specified group or all of them if `<parameter>` is *`NULL`*
`EPI_HGDS_GET_MEMBER`	extracts the specified member or all of them if `<parameter>` is *`NULL`*

- **EPIResult NMSHGDSAPIRecvReply**(
  EPIInterface api,
  EPIHGDSAPI_HostGroup * hostgroup,
  EPITimeout timeout); (C)
  EPIResult NMSHGDSAPI::**recvReply**(
  EPIHGDSAPI_HostGroup * hostgroup,
  EPITimeout timeout
  = **EPI_TIMEOUT_FOREVER**); (C++)
  Enhances `NMSAPIRecvAPIReply/NMSAPI::recvAPIReply` to automatically extract the Passport host information if present. The extracted fields are returned into a static area and a pointer to that area is returned as the `<hostgroup>` argument. This static area of type `EPIHGDSAPI_HostGroup` contains the following fields whose values must not be modified nor freed (they are valid until the next API related function call):

  *char ** **name**;
  is the Passport host or group name.

  *char ** **ipAddr**;
  is the Passport host IP address, if applicable., and

  *char ** **type**;
  is "PASSSPORT" for a host, and "GROUP" for a group.

  A timeout can be specified. With a timeout of `EPI_TIMEOUT_POLL(0)`, the command acts as a no-wait poll. A value of `EPI_TIMEOUT_FOREVER (-1)` waits forever. Other values wait for the specified number of seconds.

  Example (C)

```
if ( NMSHGDSAPISendQuery(api, EPI_HGDS_GET_CHILDREN,
            groupName) == EPI_SUCCESS ) {
  while ( NMSHGDSAPIRecvReply(api, hg,
            EPI_TIMEOUT_FOREVER) == EPI_SUCCESS) {
    if (NMSAPIGetAPIRecord(api) == RESPONSE) {
      sprintf(cmd, "/usr/sbin/ping %s",
            hg->ipAddress);
      if (system(cmd) == 0) {
            ... /* Passport is reachable */
```

Example (C++)

```
if ( api->sendQuery(EPI_HGDS_GET_CHILDREN,
                groupName) == EPI_SUCCESS ) {
  while ( api->recvReply(hg) == EPI_SUCCESS) {
    if (api->getAPIRecord() == RESPONSE) {
       sprintf(cmd, "/usr/sbin/ping %s",
                hg->ipAddress);
       if (system(cmd) == 0) {
                ... // Passport is reachable
```

- **EPIHGDSAPI_HostGroup** * **NMSHGDSAPIExtractHostGroup**(**EPIInterface** api); (C)
  EPIHGDSAPI_HostGroup * **NMSHGDSAPI**::**extractHostGroup**(); (C++)

  Extracts the Host Group information similarly to `NMSHGDSAPIRecvReply/NMSHGDSAPI::recvReply` from the last received reply be it from a call to `NMSAPIRecvAPIReply/ NMSAPI::recvAPIReply` or in a bound API Callback. The extracted fields, which must not be modified nor freed, are returned as a pointer to a static area valid until the next API related function call.

Command access

Like the `cmccmd` utility, Command Access allows programs to connect to Passport Groups and DPN-100 OAs (the command route), send commands to the modules they contain, and receive the replies. Command Access also provides several utility commands to assist with the parsing and identification of the command output. Command Access commands further communicate with the Command Session servers (CMCFUN and CM) that correspond to the current `DISPLAY` environment variable (for example, for a program that uses the session servers in the current Preside Multiservice Data Manager (MDM) User Session and potentially uses its current group and OA connections) in the Command Console.

For stand-alone (for example, CRON) or specific programs (for example, using Passport provisioning mode), it may be necessary to create a private Command Session for the execution of the program. Command Access provides the `NMSCmdSession` comand to start (and stop) such a session. See 241-6001-301 *Preside MDM Customization Administrator Guide* for more information on how to use cmcwrap to write macros).

Command Access commands are used in the following sequence:

1 Initialize a command interface.

2 Connect to the Command Session server (CMCFUN).

3 Connect to one or more Passport Group(s) or OA(s), if necessary.

4 Send a command to a node in the connected Groups and OAs.

5 Receive the command replies, either as a single string or one line at a time.

6 Disconnect from the command server.

7 Drop the command interface.

Multiple commands can be sent to the nodes, but individual command interfaces will handle these commands synchronously, one at a time. It is possible, however, to create multiple command interfaces (each one resulting in its own connection to the server and therefore capable of independent, parallel commands).

Command Access commands

The following Command Access commands are provided:

- **NMSCmdStartSession(**
  char * display ,
  char * dpnAccessHost ,
  char * ppAccessHost,
  int idletime) (C)

  NMSCmdStopSession(
  char * display) (C)

  Starts (`NMSCmdStartSession`) or stops (`NMSCmdStopSession`) a private command session for the specified `<display>` name. The specified `<display>` name must be a unique value workstation-wide for private sessions. If you use `NULL`, "`:0.0`" is used by default. `NULL` is equivalent to spawning the following command in the background:

```
/bin/env DISPLAY=<DISPLAY> \
   /opt/MagellanNMS/bin/loop
          -delay 3 \
               /opt/MagellanNMS/bin/icm \; \
               /opt/MagellanNMS/bin/cmcfun
```

When you use `loop`, the session terminates automatically within 30 seconds if the program that has spawned it terminates without stopping it. Ensure that you watch for middle shell processes. See 241-6001-301 *Preside MDM Customization Administrator Guide* for more information on the `loop` utility.

The calling program waits for two to three seconds after invoking the `loop` command before attempting to connect to the session. The waiting period lets the session servers initialize themselves properly. The calling program can also perform the operations it wants to with the same effect.

If one of `<dpnAccessHost> and/or <ppAccessHost>` are non `NULL`, the matching service selection is applied to the new session. The corresponding servers are used instead of the current workstation service selection (`NMSCmdSetServiceSelection` lets you change the session's service selection).

If `<idletime>` is non-zero, this time (in minutes) is passed to cmcfun. Cmcfun self-terminates automatically if the specified minutes elapses with no command activity. This ensures that device connection resources are not used for an extended period of time. The next time the session is used, the required group connection needs to be re-created.

Example(C)

```
/*start a private command session and wait for it
 * to initialize */
sprintf(mysession, "mysession.%d", getpid());
if (NMSCmdStartSession(mysession, NULL, NULL, 0)
              != EPI_SUCCESS) {
   exit(1);
}
sleep(3);
/* initialize an connect the command interface
 * with the named session display (see below) */
```

Note: Since this call invokes the `fork`/`exec` system calls, they create a child process to the calling program. The child process may terminate resulting into a `SIGCHLD` signal being received by the program. Make

sure you handle or ignore the signal to avoid the creation of zombie processes. EPI does not need to know about the signal to function properly. The starting of the Command Session is stateless.

- **EPIInterface** cmd = **NMSCmdInit**(); (C)
  NMSCmd::**NMSCmd**(); (C++ constructor)
  Initializes a command interface.

- *void* **NMSCmdSetUserData**(
  EPIInterface api,
  void * userData); (C)
  void **NMSCmd**::**setUserData**(
  void * userData); (C++)
  void * **NMSCmdGetUserData**(
  EPIInterface api); (C)
  void * **NMSCmd**::**getUserData**(); (C++)
  void **NMSCmdSetName**(
  EPIInterface api,
  char * name); (C)
  void **NMSCmd**::**setName**(
  char * name); (C++)
  char * **NMSCmdGetName**(
  EPIInterface api); (C)
  char * **NMSCmd**::**getName**(); (C++)
  These routines allow you to associate and extract arbitrary user data and a name to the command interface.

- *void* **NMSCmdDrop**(
  EPIInterface cmd); (C)
  NMSCmd::~**NMSCmd**(); (C++ destructor)
  Drops the command interface (frees allocated resources).

- **EPIResult NMSCmdConnect**(
  EPIInterface cmd,
  char * display); (C)
  EPIResult NMSCmd::**connect**(
  char * display = *NULL*); (C++)
  Connects the interface to the Command Session servers that correspond to the specified `<display>` variable. If `<display>` is *`NULL`*, the current

value of the `$DISPLAY` environment variable is used (suitable for interactive programs used within an Preside Multiservice Data Manager (MDM) user session or as argument to `cmcwrap`).

Example (C)

```
EPIInterface cmd = NMSCmdInit();
if ( NMSCmdConnect(cmd) == EPI_SUCCESS ) {
   ... /* we're connected to the default session */
```

Example (C++)

```
EPICmd * cmd = new NMSCmd();
if ( cmd->connect(displayStr) == EPI_SUCCESS ) {
   ... // we're connected to the alternate session
```

- **EPIResult NMSCmdDisconnect**(
  EPIInterface cmd); (C)
  EPIResult NMSCmd::**disconnect**(); (C++)
  Disconnects the interface from the session servers. The interface may be reconnected to the servers from the same session or to servers from another session.

- **EPIResult NMSCmdIsOK**(
  EPIInterface cmd); (C)
  EPIResult NMSCmd::**isOK**(); (C++)
  Returns current connection status of the interface (`EPI_SUCCESS`, `EPI_FAILED`, or `EPI_NO_CONNECTION`).

- **EPIResult NMSCmdSetServiceSelection**(
  EPIInterface cmd,
  char * dpnAccessHost,
  char * ppAccessHost) (C)
  EPIResult NMSCmd::setServiceSelection(
  char * dpnAccessHost = *NULL*,
  char * ppAccessHost = *NULL*) (C++)

  Controls the Service Selection settings for the Command Session this interface is connected to. `dpnAccessHost` specifies the MDM host name for DPN Network Access. `ppAccessHost` specifies the MDM host name for Passport Network Access. If the value of these parameters is NULL then no change is made to the corresponding Service Selection. This command is similar to using the Service Selection tool on the same session as the Session Servers.If other users (or programs/scripts) are

using that session, then they will also be impacted by this change. The Command Interface must be connected, see `NMSCmdConnect`) for this function to work.

- **EPIResult NMSCmdSendConnect**(
  EPIInterface cmd,
  char * routeType,
  char * routeName,
  char * userId,
  char * password); (C)
  EPIResult NMSCmd::**sendConnect**(
  char * routeType,
  char * routeName,
  char * userId,
  char * password); (C++)
  Sends a connect request to the indicated route using the specified authentication information. Note that this is not a combo call. The reply from this message must be collected with `NMSCmdRecvReply/ NMSCmd::recvReply` (and the likes). For your convenience, the following route type constants are defined (only the first two can be connected to, the others can be used in the other functions applicable to this interface type):

EPICMD_OA_ROUTE	(“OA”)	for DPN-100 NCS OAs
EPICMD_PP_GROUP_ROUTE	(“GROUP”)	for Passport groups
EPICMD_PP_WILD_ROUTE	(“*”)	for the special Passport wild-card route
EPICMD_MACRO_ROUTE	(“$”)	for Unix Macros
EPICMD_SNMP_ROUTE	(“@”)	for the SNMP Command framework

Note: Connecting to an already connected route results in an error, even though the route is available. Send a `“”` command to the route to verify if a connection has already been established (the reply indicates an error if the connection does not exist).

Example (C)

```
if (   (NMSCmdSendCommand(cmd, EPICMD_OA_ROUTE, "")
              != EPI_SUCCESS)
    || (NMSCmdSkipRestOfReply(cmd) != EPI_SUCCESS) )
   if ( NMSCmdSendConnect(cmd, EPICMD_OA_ROUTE,
              "myOA", "myCap", "aq1sw2")
              == EPI_SUCCESS) {
       char * txt;
       if ( NMSCmdRecvFullReply(cmd, &txt)
              == EPI_SUCCESS) {
       ...    /* we're connected to the route */
```

Example (C++)

```
if (cmd->sendCommand(EPICMD_OA_ROUTE, "")
              != EPI_SUCCESS)
   if ( cmd->sendConnect(EPICMD_OA_ROUTE,
              "myOA", "myCap", "aq1sw2")
              == EPI_SUCCESS) {
       char * txt;
       if ( cmd->recvFullReply(&txt)
              == EPI_SUCCESS) {
       ...    // we're connected to the route
```

Note: If the script is to be invoked from the Command Console, you can use *getenv*("CMC_CURRENT_ROUTE") as the destination value here and in NMSCmdSendCommand/NMSCmd::sendCommand to indicate the current Command Console route.

- **EPIResult NMSCmdSendDisconnect**(
  EPIInterface cmd,
  char * routeName); (C)
  EPIResult NMSCmd::**sendDisconnect**(
  char * routeName); (C++)
  Disconnects the interface from the named route.

- **EPIResult NMSCmdSendCommand**(
  EPIInterface cmd,
  char * routeName,
  char * command); (C)
  EPIResult NMSCmd::**sendCommand**(
  char * routeName,

char * command); (C++)

Sends a command to a node through the specified connected route. The name of the destination node is typically the first token of the command.

Example (C)

```
if ( NMSCmdSendCommand(cmd, "myOA", "R78 d")
                == EPI_SUCCESS ) {
   ... /* command sent */
```

Example (C++)

```
if ( cmd->sendCommand("myOA", "R78 d")
                == EPI_SUCCESS ) {
   ... // command sent
```

Note: The special command routes of the Command Console ($ for macro access, @ for the SNMP Command Framework, and * for the Passport wild-card group) can all be used as routes from EPI.

- **EPIResult NMSCmdRecvFullReply**(
  EPIInterface cmd,
  char ** text); (C)
  EPIResult NMSCmd::**recvFullReply**(
  char ** text); (C++)

  Waits for and receives the complete reply to the previous node command. A pointer to the replied text is returned as the `<text>` argument (do not modify nor or free this buffer). The received text is also available to the reply manipulation functions (pattern matching, column extraction, ...).

  Example (C)

  ```
  NMSCmdSendCommand(cmd, "myOA", "dir");
  char * txt;
  if ( NMSCmdRecvFullReply(cmd, &txt) == EPI_SUCCESS ) {
      if ( strstr(txt, "pe") != NULL ) {
  ...
  ```

  Example (C++)

  ```
  cmd->sendCommand("myOA", "dir");
  char * txt;
  if (cmd->NMSCmdRecvFullReply(&txt) == EPI_SUCCESS) {
      if ( strstr(txt, "pe") != NULL ) {
  ...
  ```

- **EPIResult NMSCmdRecvNextLine**(
  EPIInterface cmd,
  char ** text,
  EPITimeout timeout); (C)
  EPIResult NMSCmd::**recvNextLine**(
  char ** text,
  EPITimeout timeout
  = **EPI_TIMEOUT_FOREVER**); (C++)
  Waits for and receives the next line of the last reply of the issued node command. A timeout can be specified. With a timeout of `EPI_TIMEOUT_POLL` (`0`), the command acts a no-wait poll. With `EPI_TIMEOUT_FOREVER` (-1), it blocks waiting for a reply. Other values wait for the specified number of seconds. A pointer to the replied text is returned in `<text>`. This text must not be modified nor freed. The received text is also available to the `NMSCmdPatternMatch/ NMSCmd::patternMatch`, `NMSCmdGetNumColumns/ NMSCmd::getNumColumns`, and `NMSCmdGetColumn/ NMSCmd::getColumn` functions. The previous example can be rewritten as follows:

  Example (C)

```
NMSCmdSendCommand(cmd, “myOA”, “r78 d”);
char * line;
while ( NMSCmdRecvNextLine(cmd, &line,
                EPI_TIMEOUT_FOREVER) == EPI_SUCCESS) {
   if ( strstr(line, “pe”) != NULL ) {
...
```

  Example (C++)

```
cmd->sendCommand(“myOA”, “r78 d”);
char * line;
while ( cmd->recvNextLine(&line) == EPI_SUCCESS) {
   if ( strstr(line, “pe”) != NULL ) {
...
```

- **EPIResult NMSCmdRecvNextChunk**(
  EPIInterface cmd,
  char ** text,
  EPITimeout timeout); (C)
  EPIResult NMSCmd::**recvNextChunk**(
  char ** text,

EPITimeout timeout
= **EPI_TIMEOUT_FOREVER**); (C++)

Waits for and receives the next chunk of the last reply of the issued node command. A chunk is most efficiently processed by EPI and may contain multiple lines of text (it may even end in the middle of a line). A timeout can be specified. With a timeout of `EPI_TIMEOUT_POLL (0)`, the command acts a no-wait poll. With `EPI_TIMEOUT_FOREVER` (-1), it blocks waiting for a reply. Other values wait for the specified number of seconds. A pointer to the replied text is returned in `<text>`. This text must not be modified nor freed. The received text is also available to the `NMSCmdPatternMatch/NMSCmd::patternMatch`, `NMSCmdGetNumColumns/NMSCmd::getNumColumns`, and `NMSCmdGetColumn/NMSCmd::getColumn` functions. The previous example can be rewritten as follows:

Example (C)

```
NMSCmdSendCommand(cmd, "myOA", "r78 d");
char * line;
while ( NMSCmdRecvNextChunk(cmd, &line,
                EPI_TIMEOUT_FOREVER) == EPI_SUCCESS) {
    if ( strstr(line, "pe") != NULL ) {
...
```

Example (C++)

```
cmd->sendCommand("myOA", "r78 d");
char * line;
while ( cmd->recvNextChunk(&line) == EPI_SUCCESS) {
    if ( strstr(line, "pe") != NULL ) {
...
```

- **EPIResult NMSCmdSkipRestOfReply**(
  EPIInterface cmd,
  int * error); (C)

  EPIResult NMSCmd::**skipRestOfReply**(
  int * error = *NULL*); (C++)

  This combination call waits for and ignores all replies until the end of response. If `<error>` is non-*NULL* and a processing error is detected while receiving replies (not an on-switch error indication), then `<error>` will be set to `1`.

- *int* nbCols = **NMSCmdGetNumColumns**(
  EPIInterface cmd); (C)
  int nbCols = **NMSCmd**::**getNumColumns**(); (C++)
  Returns the number of blank (space or tab) separated columns in the previously received reply line.

- **EPIResult NMSCmdGetColumn**(
  EPIInterface cmd,
  char ** text,
  int colNo); (C)
  EPIResult NMSCmd::**getColumn**(
  char ** text,
  int colNo); (C++)
  Returns a pointer to the indicated. `<colNo>` (starting from `1`), blank (space or tab) separated column of the previously received reply line in `<text>`. The returned text must not be modified nor deleted.

  Example (C)

```
NMSCmdSendCommand(cmd, “myPassp”, “pp1 d fruni/101”);
char * col, * dontcare;
while ( NMSCmdRecvNextLine(cmd, &contcare)
                == EPI_SUCCESS ) {
   NMSCmdGetColumn(cmd, &col, 1);
   strcat(att, col);
   nb = NMSCmdGetNbColumns(cmd);
   for (i = 3; i < nb; i++) {
              NMSCmdGetColumn(cmd, &col, i);
              strcat(val, col);
   }
   if { strcasecmp(att, “operationalState”) == 0 ) {
      printf(“State: %s\n”, val);
...
```

  Example (C++)

```
cmd->sendCommand(“myPassp”, “pp1 d fruni/101”);
char * col, * dontcare;
while ( cmd->recvNextLine(&contcare)
                == EPI_SUCCESS ) {
   cmd->getColumn(&col, 1);
   strcat(att, col);
   nb = cmd->getNbColumns();
   for (i = 3; i < nb; i++) {
              cmd->getColumn(&col, i);
```

```
                strcat(val, col);
      }
      if { strcasecmp(att, "operationalState") == 0 ) {
         cout << "State: " << val << endl;
...
```

- **EPIResult NMSCmdPatternMatch**(
  EPIInterface cmd,
  char ** string,
  char * pattern,
  char * substitute,
  int all); (C)

  EPIResult NMSCmd::**patternMatch**(
  char ** string,
  char * pattern,
  char * substitute = *NULL*,
  int all = 0); (C++)

  This is the functionality of `NMSEPIPatternMatch` applied to the last received command response (full, line, or chunk). The difference is that it returns its success indication as a returned value and a pointer to the matched or substituted portion through the `<string>` pointer (do not modify nor free this value).

  Example (C)

```
NMSCmdSendCommand(cmd, "myOA", "dir");
char * line;
while ( NMSCmdRecvNextLine(cmd, &line)
                == EPI_SUCCESS ) {
   char * dontcare;
   if ( NMSCmdPatternMatch(cmd, &dontcare, ".*pe.*",
                NULL, 0) == EPI_SUCCESS ) {
                puts(line);
...
```

  Example (C++)

```
cmd->sendCommand("myOA", "dir");
char * line;
while ( cmd->recvNextLine(&line) == EPI_SUCCESS ) {
   char * dontcare;
   if ( cmd->patternMatch(&dontcare, ".*pe.*")
```

```
                                    == EPI_SUCCESS ) {
                        cout << line << endl;
    ...
```

- **EPIResult NMSCmdSendListRequest**(
  EPIInterface cmd,
  char * routeType); (C)
  EPIResult NMSCmd::**sendListRequest**(
  char * routeType); (C++)
  This special utility command requests a list of all OA, Passport Group, or both) types of available routes (when `<routeType>` set to `EPICMD_OA_ROUTE` (“OA”), `EPICMD_PP_GROUP_ROUTE` (“GROUP”), or `EPICMD_ALL_ROUTE` (“ALL”) respectively). It corresponds to the `cmccmd list` command and is similar in behavior to an `NMSCmdSendCommand/NMSCmd::sendCommand` function call. The replies can be extracted with `NMSCmdRecvNextLine/ NMSCmd::recvNextLine` and have the following structure:
  `<name> <type> <state>`
  where `<name>` is the destination name, `<type>` is its type (`OA` or `GROUP`), and `<state>` its connection state (`CONN`, for connected, `AUTH`, for authentication, and `-` otherwise)

  Example(C)

```
NMSCmdSendDestRequest(cmd, “GROUP”);
char * dontcare;
while ( NMSCmdRecvNextLine(cmd, &dontcare)
                == EPI_SUCCESS ) {
    char * col;
    if (   (NMSCmdGetColumn(cmd, &col, 2)
                == EPI_SUCCESS)
        && (strcasecmp(col, “CONN”) == 0) ) {
                ... # route is available
```

  Example(C++)

```
cmd->sendDestRequest(“GROUP”);
char * dontcare;
while (cmd->recvNextLine(&dontcare) == EPI_SUCCESS) {
    char * col;
    if (   (cmd->getColumn(&col, 2) == EPI_SUCCESS)
        && (strcasecmp(col, “CONN”) == 0) ) {
                ... # route is available
```

- *void* **NMSCmdBindCallback**(
  EPIInterface cmd,
  EPICallBackFunction aCbFn); (C)
  void **NMSCmd**::**bindCallback**(
  EPICallBackFunction aCbFn); (C++)
  Binds the specified callback function to the Command interface. This callback will be invoked whenever a new message is received from the server for this interface and the program is in an event loop (`NMSEPIEventLoop`). The callback mechanism acts as though the indicated function was called immediately after a call to `NMSCmdRecvNextLine/NMSCmd::recvNextLine` (by default), or `NMSCmdRecvNextChunk/NMSCmd::recvNextChunk` (if `NMSCmdSetAsyncByChunk/NMSCmd::setAsyncByChunk` is called with `1`), or `NMSCmdRecvNextPPComp/NMSCmd::recvNextPPComp`(if `NMSCmdSetAsyncByPPComp/NMSCmd::setAsyncByPPComp` is called with 1) . The callback signature is:

  `typedef` *void (**`EPICallBackFunction`*) (void ** cmd, *char ** text, *int* notused, *int* status*)*

  where `<cmd>` is an opaque pointer to the calling Command interface (as returned by `NMSCmdInit/NMSCmd::NMSCmd`), `<text>` is the received text line or chunk (or *`NULL`* if not applicable), and `<status>` is one of `EPI_SUCCESS`, for a successful reception, `EPI_END` to indicate an end of response, `EPI_NO_CONNECTION`, if the connection to the server was lost, and `EPI_FAILED`, for other error cases. The special value `EPI_FLOW_CB` indicates a callback invoked by a `@CB` construct in a command flow (`NMSCmdDoCommandFlow`/`NMSCmd::doCommandFlow` or `NMSCmdDoCommandFile`/`NMSCmd::doCommandFile`).

  In addition, the post-`Recv` functions (i.e., `NMSCmdGetColumn/ NMSCmd::getColumn`, `NMSCmdPatternMatch/ NMSCmd::patternMatch`, ...) can be used in the context of this callback to manipulate the received reply. In Passport component mode, the Passport information extraction functions (`NMSCmdGetPPCompID/ NMSCmd::getPPCompID, NMSCmdResetPPCompAttrs/ NMSCmd::restPPCompAttrs,...)` can also be used. The event loop is launched with `NMSEPIEventLoop` and never returns.

Example (C)

```
void
mycb (cmd, text, notused, status)
void * cmd;
char * text;
int notused;
int status;
{
   /* ...Process the reply... */
}
...
NMSCmdSendCommand(cmd, "myOA", "r72 q serv");
NMSCmdBindCallback(cmd, mycb);
...
NMSEPIEventLoop(); /* never returns */
```

Example (C++)

```
void
mycb (void * cmdP, char * text, int, int status)
{
   NMSCmd * cmd = (NMSCmd*)cmdP;
   // ...Process the reply...
}
...
cmd->sendCommand("myOA", "r72 q serv");
cmd->bindCallback(mycb);
...
NMSEPIEventLoop(); // never returns
```

- *void* **NMSCmdSetAsyncByChunk**(
  EPIInterface cmd,
  int onoff); (C)
  void **NMSCmd**::**setAsyncByChunk**(
  int onoff = 1); (C++)

  void **NMSCmdSetAsyncByPPComp**(
  EPIInterface cmd,
  int onoff); (C)
  void **NMSCmd**::**setAsyncByPPComp**(
  int onoff = 1); (C++)
  The `NMSCmdSetAsyncByChunk/NMSCmd::setAsyncByChunk` function informs the Command interface that asynchronous results

returned through a bound callback should be provided one line at a time (`onoff` set to `0`, the default behavior), or a chunk at a time (`onoff` set to `1`).
The function `NMSCmdSetAsyncByPPComp/NMSCmd::setAsyncByPPComp`, on the other hand, informs the Command interface that the asynchronous results should be provided one Passport component at a time (see `NMSCmdRecvNextPPComp/NMSCmd::recvNextPPComp`).
Both functions must be called after the callback is set with `NMSCmdBindCallback/NMSCmd::bindCallback`.

- **EPIResult NMSCmdRecvNextPPComp**(
  EPIInterface cmd,
  EPITimeout timeout); (C)
  EPIResult NMSCmd::**recvNextPPComp**(
  EPITimeout timeout
  = **EPI_TIMEOUT_FOREVER**); (C++)
  `NMSCmdRecvNextPPComp` is a form of receive function that waits for and receives the next Passport component reply to the previous command (i.e. the result of a `list` or `display` command). A timeout can be specified. With a timeout of `EPI_TIMEOUT_POLL (0)`, the command acts as a no-wait poll. A value of `EPI_TIMEOUT_FOREVER (-1)` waits forever. Other values wait for the specified number of seconds.

  Note: Make sure you always specify the `-notab` (no tabular output) CAS command line option when sending a Passport CAS display command with wildcards if this command is to be used to extract the replies.

  The `NMSCmdGetPPCompID/NMSCmd::getPPCompId`, `NMSCmdResetPPCompAttrs/NMSCmd::resetPPCompAttrs`, `NMSCmdGetFirstPPCompAttr/NMSCmd::getFirstPPCompAttr`, `NMSCmdGetNextPPCompAttr/NMSCmd::getNextPPCompAttr`, and `NMSCmdFindNextPPCompAttr/NMSCmd::findNextPPCompAttr` commands can then be used to extract the replied component information.

- *char* * compid = **NMSCmdGetPPCompID**(
  EPIInterface cmd); (C)
  char * compid = **NMSCmd**::**getPPCompID**(); (C++)

`NMSCmdGetPPCompID` extracts the name of the last received Passport component. The list of Passport component attribute is also reset to the beginning for the `NMSCmdGetNextPPCompAttr/NMSCmd::getNextPPCompAttr` and `NMSCmdFindNextPPCompAttr/NMSCmd::findNextPPCompAttr` commands.

Example (C)

```
NMSCmdSendCommand(cmd, "myGroup", \
   "TOTO display shelf card/* utilization");
while ( NMSCmdRecvNextPPComp(cmd,
              EPI_TIMEOUT_FOREVER) == EPI_SUCCESS ) {
   char * compid = NMSCmdGetPPCompID(cmd);
   printf("Card: %s\n", compid);
...
```

Example (C++)

```
cmd->sendCommand("myGroup", \
   "TOTO display shelf card/* utilization");
while ( cmd->recvNextPPComp() == EPI_SUCCESS ) {
   char * compid = cmd->getPPCompID();
   cout << "Card: " << compid << endl;
...
```

- *void* **NMSCmdResetPPCompAttrs**(
  EPIInterface cmd); (C)
  void **NMSCmd**::**resetPPCompAttrs**(); (C++)
  `NMSCmdResetPPCompAttrs` simply resets the list of Passport component attributes received by the last call to `NMSCmdRecvNextPPComp/NMSCmd::recvNextPPComp` (or asynchronous callback in Passport component mode) to the beginning for the `NMSCmdGetNextPPCompAttr/NMSCmd::getNextPPCompAttr` and `NMSCmdFindNextPPCompAttr/NMSCmd::findNextPPCompAttr` commands.

- **EPICmdPPAttr** * attr = **NMSCmdGetFirstPPCompAttr**(
  EPIInterface cmd); (C)
  EPICmdPPAttr * attr = **NMSCmd**::**getFirstPPCompAttr**(); (C++)
  `NMSCmdGetFirstPPCompAttr` extracts the first attribute from the last received Passport component. The attribute is returned as a pointer to a static structure whose contents must not be modified nor freed. This

structure contains the following fields:

char * **name**;
the attribute name, and

char * **value**;
its value.

If the attribute is a list, vector or array, the first index is added to the name with a coma as separator (for example, "`pktFromIfByPrio,ep0`"). For two dimensional arrays, an entry is created with the attribute name and the column title as value. Another entry is created with "`<attribute name>,<row title>`" as a name and the list of column labels as value. The remaining entries for this attribute have "`<attribute name>,<row label>`" as name and the list of corresponding columns as values. Finally, the following special attributes are also available; `"Message" (name),` contains as value any message emitted by Passport not part of an attribute value (i.e. error messages), `"CompID" (name)` has for value the returned component's name.

Example (C)

```
...
EPICmdPPAttr * attr;
while ( attr = NMSCmdGetFirsttPPCompAttr(cmd) ) {
    printf("Attribute: %s\n", attr->name);
    printf("Value: %s\n", attr->value);
...
```

Example (C++)

```
...
EPICmdPPAttr * attr;
while ( attr = cmd->getFirstPPCompAttr() ) {
    cout << "Attribute: " << attr->name << endl;
    cout << "Value: " << attr->value << endl;
...
```

- **EPICmdPPAttr** * attr = **NMSCmdGetNextPPCompAttr**(
  EPIInterface cmd); (C)
  EPICmdPPAttr * attr = **NMSCmd**::**getNextPPCompAttr**(); (C++)

Same as `NMSCmdGetFirstPPCompAttr/ NMSCmd::getFirstPPCompAttr` but extracts the next attribute rather than the first one.

- **EPICmdPPAttr** * attr = **NMSCmdFindNextPPCompAttr**(
  EPIInterface cmd,
  char * attrName); (C)
  EPICmdPPAttr * attr = **NMSCmd**::**findNextPPCompAttr**(
  char * attrName); (C++)
  `NMSCmdFindNextPPCompAttr` extracts the next attribute from the last received Passport component whose name (see the description of the `EPICmdPPAttr` structure in `NMSCmdRecvNextPPComp/ NMSCmd::getFirstPPCompAttr above)` matches the specified `<attrName>` value.

  Example (C)

```
...
long long prev;
EPICmdPPAttr * attr;
if ( attr = NMSCmdFindNextPPCompAttr(cmd,
              "bytesSent") ) {
    long long val = atoll(attr->value);
    printf("Delta: %ull\n", prev - val);
    prev = val;
...
```

  Example (C++)

```
...
long long prev;
EPICmdPPAttr * attr;
if ( attr = cmd->findNextPPCompAttr("bytesSent") ) {
    long long val = atoll(attr->value);
    cout << "Delta: " << (prev - val) << endl;
    prev = val;
...
```

- **EPIResult** res = **NMSCmdDoCommandFile**(
  EPIInterface cmd,
  char * filePath,
  char * dest, **EPIAVL** * avl,
  char ** output, *int* *result,
  int bestEffort, *int* trace, *int* test,

```
    int cb); (C)
EPIResult res = NMSCmd::doCommandFile(
    char * filePath,
    char * dest, EPIAVL * avl,
    char ** output, int *result,
    int bestEffort, int trace, int test,
    int cb); (C++)

EPIResult res = NMSCmdDoCommandFlow(
    EPIInterface cmd,
    char * commandFlow,
    char * dest, EPIAVL * avl,
    char ** output, int *result,
    int bestEffort, int trace = 0, int test = 0,
    int cb = 0); (C)
EPIResult res = NMSCmd::doCommandFlow(
    char * commandFlow,
    char * dest, EPIAVL * avl,
    char ** output, int *result,
    int bestEffort, int trace = 0, int test = 0,
    int cb = 0); (C++)

EPIAVL * avl = NMSCmdBuildAVL(
    EPIAVL * avl,
    char * name,
    char * value, ... /* NULL */);
void NMSCmdFreeAVL(EPIAVL * avl);

int res = NMSCmdTerminateCommandFlow(
    EPIInterface cmd,
    int result); (C)
int res = NMSCmd::terminateCommandFlow(
    int result = 0); (C++)
```

These calls support the synchronous execution (commands and replies) of command flows from the file identified by `<filePath>` or the string identified by `<commandFlow>`. A command flow is a sequence of device commands (one per line) to be executed as one. The file may be specified as a formal path starting with '`/`', '`.`' or '`..`', or '`~`' (which automatically

substituteso the user's HOME account). The file may also be specified as a relative path and is searched for in $HOME/MagellanNMS/<file path>, then /opt/MagellanNMS/cfg/<file path>, and finally /opt/MagellanNMS/lib/<file path>.

All commands are executed through the named destination (if empty -- "" or NULL --, the destination must prefix every device command in the flow). The execution of the flow terminates at the first failure unless <bestEffort> is set to a nonzero value. Variable substitution on each command in the flow is supported with the attribute-value pairl ist specified by <avl> as the source of the variable name-value pairs. avl is a NULL terminated array of EPIAVL structures containing the following members:

*char * **name***	substitution variable name
*char * **value***	substitution variable value

To help create the AVL, two functions are provided. NMSCmdBuildAVL takes an existing AVL as first argument (NULL to create a new one), followed by a NULL terminated list of attribute name and value strings. The new attributes are merged into the existing AVL. If a named attribute is already in the list, its value is replaced by the new one. An AVL must be freed with NMSCmdFreeAVL when it is not required.

Example (C/C++)

```
...
EPIAVL * avl = NMSCmdBuildAVL(NULL, "atmif", "112",
                              "vpci", "1.12", NULL);
... execute a flow commands
avl = NMSCmdBuildAVL(avl, "vpci", "1.13",
                              "qos", "gold", NULL);
... more flow commands
NMSCmdFreeAVL(avl); avl = NULL;
...
```

The flow may produce some output text and numerical results which are returned as <output> and <result> respectively (the output text must not be modified nor freed). The output text is produced from the flow using the @PRINT directive (see below). If <trace> is set to a nonzero

value, the executed device commands are also added to the output text (with a `#*` prefix for plain commands and `#?` for conditional commands followed by their output and a `#? END`, `#? FAILED`, `#* END`, or `#* FAILED` line to indicate the end of the corresponding command). The following constants can be ORed together to produce the trace mask:

EPICmd_TRACE_NONE	(0)	no automatic command/response tracing (default)
EPICmd_TRACE_ALL	(1)	traces all commands, controls, and responses
EPICmd_TRACE_COMMAND	(2)	traces plain commands (non `@SWITCH`/`@FOREACH`)
EPICmd_TRACE_RESPONSE	(4)	traces all plain command responses
EPICmd_TRACE_ERRRESP	(8)	traces only responses of failed commands
EPICmd_TRACE_CONTROL	(16)	traces commands and responses for control commands (`@SWITCH`, `@FOREACH`)

The output `<result>` is produced from the flow by the `@EXIT` directive or the `NMSCmdTerminateCommandFlow`/`NMSCmd::terminateCommandFlow` function in callback mode (see below). The output processing command (`NMSCmdGetNumColumns`/`NMSCmd::getNumColumns`, `NMSCmdGetColumn`/`NMSCmd::getColumn`, and `NMSCmdPatternMatch`/`NMSCmd::patternMatch`) can also be used to examine the output. If `cb` is set to a nonzero value, the output text is not returned in the `output` variable. Instead, the callback bound to the command interface by `NMSCmdBindCallback`/`NMSCmd::bindCallback` is invoked for each line of output (including the `@PRINT` and `#?` and `#*` comments) where it is available in the usual manner (with reason `RESPONSE`). From the callback, it is possible to force-terminate the execution of the flow by

invoking the `NMSCmdTerminateCommandFlow`/ `NMSCmd::terminateCommandFlow` function with the desired numerical result (as if `@EXIT` had been reached in the file).

Note that in callback mode, even though the output is returned line by line through the bound callback, the execution of the flow is still synchronous and no other EPI actions will be performed until the flow execution has completed. In other words, the flow/file execution function will not return until the flow execution is complete during which the bound callback will be invoked for each output line. Once the flow execution completes, the bound callback is called with reason `ENDRESP` on success or ERROR on failures, unless the execution was explicitly terminated with `NMSCmdTerminateCommandFlow`/ `NMSCmd::terminateCommandFlow`.
If `test` is set to a nonzero value, the command flow/file is executed in test mode as described later.

Command flows consist of a list of device commands with, optionally, substitution variables identified by a dollar sign '$'. To specify a plain $, escape it as '\$'. For example, the following is a Passport command to set the <committedInformationRate> of a FrameRelay DLCI (read as one line):

```
$name set FrUni/$fruni Dlci/$dlci Sp\
          committedInformationRate $cir
```

A flow containing such a line could be executed with an attribute-value list containing at least:

```
EPIAVL * avl = {
          {“name”,  “NODER16”},
          {“fruni”, “120”},
          {“dlci”, “25”},
```

```
            {“cir”,  “56000”},
            { NULL,  NULL}
};
```

Other forms of substitution variable specification include;

${<variable name>**}**	same as without the brackets
$!**<variable name>, or **${!<variable name>**}**	(strict substitution). When you use this variable in device commands, it does not apply to special directives. The command is skipped (silently not executed) if the specified variable has no associated value or is empty. This form is only available in actual device commands.
${<variable name>**:-** <default value>**}**	If the specified variable has no associated value or is empty, the specified default value substituted.
$%	This internal variable holds the contents of the last executed `@SWITCH` command or the value of the matched `\(\)` subpattern of the last executed `@case` command.
$%%	This special internal variable holds the pattern-matched contents (whole or sub-pattern) of the last executed `@IF` command (see below).

$?	This variable contains the numerical result of the last issued macro (@DO) or flow (@INCLUDE/@RUN).
${<variable name>**[**<index>**]}**	This represents an associative array value in the Flow language itself (not to be mistaken by array values in the scripting language). Such values can be directly set (@SET, @DEFINE, @LOCAL) or provided by specialized commands (@FOREACHPP, @SPLITCOMP, @SPLIT). When used, both the variable name and the index can be substituted variables (for example, ${array[$i]} in a loop that increments the value of $i). The ! and :- constructs apply also to the array entry (for example, ${array[$i]:-0} defining 0 as the default value for the entry).

Command flows also support special processing directives as indicated in the following table:

@PRINT <string>	append the specified text (after variable substitution) to the command output.
@FORMAT <multi-line string...> **@END**	same as @PRINT but for a multi-line piece of text.
@EXIT [<code>]	terminates the flow's execution with the specified result code. If no @EXIT directive is executed, the flow will have its result code set to 0 on success or 4 if the flow was terminated by a failed command.

@RETURN [<code>]	like @EXIT but when invoked from an included file (see @INCLUDE), only terminates the execution of that file, returning to the calling flow/file.If used from within a macro or included/run flow, the return value is available as the $? variable from the calling flow.
@TRY [<command>]	executes the command and ignores its possible failure (even if <bestEffort> was set to 0). If the command is omitted, sets the current operating mode for subsequent commands as if bestEffort had been specified as nonzero. Other modifiers (@TRACE/ @NOTRACE, @LOG/@NOLOG) may also be specified.
@CRITICAL [<command>] or **@CRIT** [<command>]	executes the command and terminates the flow if the command fails even if <bestEffort> was set to a nonzero value. If the command is omitted, sets the current operating mode to critical for subsequent commands. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.
@TRACE [<command>]	traces this command even if <trace> is set to 0. If the command is omitted, sets the current operating mode to trace (as if tracing had been enabled when the C/ C++ call was made) for subsequent commands. Other modifiers (@TRY/ @CRITICAL, @LOG/@NOLOG) may also be specified.

@NOTRACE [<command>]	does not trace this command even if -<trace> is set to a nonzero value. If the command is omitted, sets the current operating mode to no-trace for the subsequent commands. Other modifiers (@TRY/@CRITICAL, @LOG/@NOLOG) may also be specified.
@LOG [<command>]	logs this command as long as a log file has been defined (NMSCmdOpenCommandLog or @LOGFILE). If the command is omitted, sets the current operating mode to logging. Other modifiers (@TRY/ @CRITICAL, @TRACE/@NOTRACE) may also be specified.
@NOLOG [<command>]	does not log this command, even if logging was enabled. If the command is omitted, sets the current operating mode to no-logging for the subsequent commands. Other modifiers (@TRY/ @CRITICAL, @TRACE/@NOTRACE) may also be specified.
@LOGFILE [[**+**]<log file path>]]	This controls command logging (see NMSCmdOpenCommandLog). Without arguments, this is equivalent to @LOG. If the log file is identified, the issued commands are logged to it from this point on (unless modified by @NOLOG). If the file path is prefixed with the plus (+) sign, the logs are appended to the file if it exists (else the file is overwritten with the new logs).

```
@IF <var.> [==|!=
           <patterns>]
or
@IF <var.> <|>|<=|>=
           <value
<commands>
...
[@ELSE
<commands>
...]
@END
```

evaluates the specified variable expressions. This command executes the first command block if it finds that it matches one (==) or does not match any (!=) of the patterns in the first form. This command also compares (numeric or string, as appropriate), the second form, to the specified value. Otherwise, the command block following the @ELSE is executed, if there are any commands. If only the variable expression is specified, the test is positive if the expanded value is not empty. If the test was a pattern matching one, the value of the sub-pattern is available as the $%% internal variable.

```
@FOR <var> <from> <to>
      [<increment>]
or
@FOR <var> IN
      <token list>
<commands>
...
@END
or
@FOR <var> KEYS
      <array name>
<commands>
...
@END
```

executes the command block repeatedly. In the first form, the command block is executed repeatedly while incrementing the named (<var>) numerical variable in the AVL from <from> to <to> in jumps of <increment> (defaults to 1). In the second form, the command block is executed repeatedly while iterating the variable across the list of blank separated tokens. You can use the variable ($<var>) in the command block. You can terminate the loop prematurely by calling the @BREAK command from within the command block. Loops can be nested. The third form allows the variable to scan the existing indicies of the named associative array (for example, the Passport attribute values from @FOREACHPP).

```
@FOREACHPP <var>
       <Passport
        list/display
        command>
<commands>
...
@END
```

executes the specified Passport list command then iterates the command over the returned component names, assigning its name to the named variable and executing the command block. You can terminate the loop prematurely by calling the `@BREAK` command from within the command block. Loops can be nested. If the Passport command was a display one, the extracted attribute values are also available as the entries of an associative array of the same name as the specified variable. They are provided in the same way as from the `NMSCmdRecvNextPPComp` function. If wild-cards are used, make sure you specify the -notab display command option.

```
@FOREACHLN <var>
       <command>
<commands>
...
@END
```

executes the specified command then iterates the command over the returned output line by line, assigning each one to the named variable and executing the command block. You can terminate the loop prematurely by calling the `@BREAK` command from within the command block. Loops can be nested.

@BREAK

invoked from within a `@FOR/ @FOREACHPP/@FOREACHLN/@WHILE/ @WHILDO` loop construct. This command terminates the enclosing loop prematurely. In other situations, the command acts like `@RETURN` with no return code.

@CONTINUE	invoked from within a @FOR/ @FOREACHPP/@FOREACHLN/@WHILE/ @WHILDO loop construct. This command causes the loop to immediately iterate to the next cycle. Like @BREAK, the following loop-code is not executed but unlike @BREAK, the loop is not abandoned.
@CB <text>	if the Flow is running in callback mode (-cb option), the callback is invoked with the specified text as the output text (NMSEPI_OUTPUT_TEXT). The callback reason (NMSEPI_CB_REASON) is then set to FLOW_CALLBACK. The callback can interpret this text as convened and, in reply, set or reset AVL variables with NMSCmdSetFlowCBAVL before returning so the flow can use the results. You can use this to query another system or a user for a value needed in the flow processing (Wizzard).

```
@SWITCH <test command>
[<commands>]
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

executes the test command and executes the commands following the first `@CASE` whose patterns match the output of the test command (the optional commands that follow the `@SWITCH` before the first `@CASE` are always executed). Only one `@CASE` block in the `@SWITCH` construct is executed. `@SWITCH` blocks can be nested. Patterns are specified in GREP format with a '`|`' separating alternatives. If no pattern is specified, the `@CASE` block accepts any output.
The `@SWITCH` command is traced to ouput, if enabled, as:

```
#? <test command>
<command output...>
#? END
```

(If the command could not be executed, the last line will be **`#? FAILED`** instead).
The `@CASE` patterns may contain one subexpression (`\(\)`), each of whose matched value is available in the following code as the `$%` substitution variable. The usual modifiers (`@TRY`/`@CRITICAL`, `@TRACE`/`@NOTRACE`, `@LOG`/`@NOLOG`) can be specified after the `@SWITCH`.

```
@SWITCHVAL <value expr.>
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

this construct behaves much like `@SWITCH` but instead of getting its pattern match target value from a device command output, that value is directly specified as a parameter.

```
@WHILE <var.>
       [==|!=|<|>|<=|>=
       <patterns/value>]
<commands>
...
@END
```

repeatedly executes the command block as long as the text (same as `@IF`) succeeds. The loop can be broken prematurely by invoking `@BREAK`.

```
@INCLUDE <file path>
or
@RUN <file path>
```

the named file is included and processed as though its content was part of the current flow. If the file is identified as a relative path, the standard Preside Multiservice Data Manager (MDM) search path applies (see above). The difference between `@INCLUDE` and `@RUN` is that variable definitions (`@DEFINE`) performed in the file invoked by `@INCLUDE` apply to the calling flow whereas those in a file invoked by `@RUN` are ignored upon return. If the file cannot be read, the flow's execution terminates unless `bestEffort` was set to a nonzero value or the `@TRY` prefix is used. The usual modifiers (`@TRY`/`@CRITICAL`, `@TRACE`/`@NOTRACE`, `@LOG`/`@NOLOG`) can be specified before `@INCLUDE`/`@RUN`.

```
@MACRO <name>
     [<parameters...>]
<code text>
...
@ENDMACRO
```

defines a new macro that can be executed later with `@DO/@IFDO/@WHILEDO`. The macro can be given a number of positional argument names to be provided when executed (the last specified parameter name gets all the remaining arguments passed). These parameter names have local scope to the macro (as if defined with `@LOCAL`). The macro can be recursive. Just like `@INCLUDE/@RUN`, it can be terminated by `@RETURN` which specifies a return code available as the `$?` variable to the caller. Variables defined/set by the macro have the same scope as the caller unless defined with `@LOCAL`. Macros must be defined before they are used. The macros themselves have global scope and can be redefined.

```
@DO <macro name>
      [<arguments...>]
```

invokes the named defined macro. The specified arguments are assigned (local scope to the macro) to the macro's parameter names -- the name gets the left over arguments). The `@RETURN`ed value from the macro is available as the `$?` internal variable.
The usual modifiers (`@TRY/@CRITICAL`, `@TRACE/@NOTRACE`, `@LOG/@NOLOG`) can be specified before `@DO`.

```
@IFDO <macro name>
     [<arguments...>]
<command>
...
[@ELSE
<commands>
...]
@END
```

merges the functionality of the @IF and @DO constructs. Executes the macro as for @DO then performs either the first command block if the macro returns a 0 result, else performs the second (@ELSE) command block if any. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @IFDO.

```
@WHILEDO <macro name>
     [<arguments...>]
<commands>
...
@END
```

merges the functionality of the @WHILE and @DO constructs. Repeatedly executes the macro as for @DO then the command block as long as the macro returns a 0 result. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @WHILEDO. As with @WHILE, the loop can be broken with @BREAK invoked in the command block.

```
@DEFINE <name> <value>
```

defines or redefines a variable name. The new value is available in the current command block and the blocks it invokes, notably for included files. For example, if a @DEFINE is invoked in a @CASE block, the modified value applies to the commands in that block but not in the commands that follow the @SWITCH construct for that @CASE. Similarly, @DEFINEs used in included files have no effect on the calling command block.

@UNDEFINE <name>	Contrary to @DEFINE, @SET, and @LOCAL, undefines the named variable. All matching associative array values are also undefined. To undefine a single entry in the array, specify its full name (for example, @UNDEFINE <array>[<index>]).
@LOCAL <name> [<value>]	Like @DEFINE but the new variable has local scope (it will not replace nor exist in the caller's scope, for example when used from within a macro or included flow). The local scope also applies to associative arrays by that name. This is useful when defining macros that need variables for which you do not want to override existing values.
@SET <name> <value1> [+|-|*|/|%|~ <value2>]	similar to @DEFINE, but this command sets the variable to the result of the numerical expression (+, -, *, /, % -- remainder). The ~ operator performs a pattern match using the second value as a GREP style pattern list (| separated). The variable is set to the matching portion or to nothing. If the pattern contains a \(\) delineated sub-pattern, it is the matching sub-pattern is used as the new value. If only the name and first value are specified, the effect is the same as @DEFINE.

```
@SPLIT[(<separators>)]
        <array name>
        <string>
```

This tokenizes the specified string. The individual tokens are assigned to indexed elements of the named associative array (starting at `1`). The actual variable's value is the number of resulting tokens. By default, tokenization is done on blanks. Alternatively, the separator characters can be provided between brackets. For example, the following will print the individual applications in a Passport AVL, one per line:

```
@SPLIT(, \t\n) app $avl
@FOR i 1 $app
     @PRINT ${app[$i]}
@END
```

@SPLITCOMP <array name> <component ID>

analyzes the specified component ID and provides the results in the named associative array. The following examples are based on the component EM/TOTO LP/2 Ds1/0.

- **$**<array> the component ID in API format (EM TOTO LP 2 DS1 0)
- **${**<array>**[_MOD]}** the module name (TOTO)
- **${**<array>**[_SUB]}** the subcomponent portion (minus first level) (LP/2 DS1/0)
- **${**<array>**[_PAR]}** the parent subcomp portion (minus last level) (EM/TOTO LP/2)
- **${**<array>**[**<category>**]}** the relative instance value for that level (EM -> TOTO, LP -> 2, DS1 -> 0)

@WAIT <nb seconds>

blocking wait for the specified number of seconds.

```
@ASK <name>
    [:E|:I|:S]
    [/validation
patterns/]
    [=<default value>]
    <prompt string>
@CASK <name>
    [:E|:I|:S]
    [/validation
patterns/]
    [=<default value>]
    <prompt string>
```

asks you (standard input) for the value of the named variable using the specified string as a prompt. The expected type of the value can be specified as one of the following for a plain string:

`:E` for a string token enumeration

`:I` for an integer

`:S` (the default)

You can use a pattern between two forward slashes (//), vertical bars (|). Specify the pattern so that EPI validates the entered value and prompts if there is no match. For enumeration, the pattern is a blank/coma separated list of words to match (for example, `/on off/`). For integers, the pattern is a blank/coma separated list of numeric values or ranges (for example, `/1,3,5,10-15,20/`). For strings (default) the pattern is an extended GREP style pattern list with `|` between alternative patterns (for example, `/.* Ds1\/.*|.* E1\/.*/` -- The forward `/` in the pattern is escaped with a single backwards `\` so it is not included as the end of the pattern). If you specify a default value, this value is set to the variable if you enter nothing (carriage return only). If you do not specify a default value, you are prompted when you enter an empty string.

`@CASK` (conditional ask) is similar to `@ASK` except that it does not prompt if the variable already has a non-empty value.

`# <comment>`	comment line.

`<device command>`	actual device command, optionally with embedded variables (`$` prefix) invoked a‘fter substitution. If the command fails, the flow execution terminates (`<bestEffort>` set to 0 or `@TRY` prefix).
	The command is traced to ouput, if enabled, as:
	`#* <device command>`
	`<command output...>`
	`#* END`
	(the last line will be `#* FAILED` instead if the command indicated an error).

Note: Note that `@TRY`, `@CRITICAL`, `@TRACE`, and `@NOTRACE` can be combined. They can also be used with the `@INCLUDE` directive. Also, the command specified with `@SWITCH` can also start with `@TRACE` or `@NOTRACE`.

Example
Assuming a file (`examp.tmpl`) containing:

```
# Frame Relay QOS example
@SWITCH $name l FrUni/$fruni Dlci/$dlci
@CASE failed|ERROR
  @PRINT $name FrUni/$fruni Dlci/$dlci does not exist!
  @EXIT 1
@CASE
  $name set FrUni/$fruni Dlci/$dlci Sp cir $cir
  $name set FrUni/$fruni Dlci/$dlci Sp bc $bc
  $name set FrUni/$fruni Dlci/$dlci Sp be $be
  @IF $lmi
    $name set FrUni/$fruni Lmi procedures $lmi
    $name set FrUni/$fruni Lmi side $lmiside
  @END
@END
```

In C, this flow can be executed with:

```
int result;
char * output;
EPIAVL * avl; /* populated as shown above or by code*/
if (NMSCmdDoCommandFile(“examp.tmpl”, “*”, avl,
                        &output, &result, 0, 0)
                != EPI_SUCCESS) {
    fprint(“Failed (%d)!!!\n%s\n”, result, output);
    exit(1)
}
```

If, instead, the contents of the file above is stored/built in a text buffer (say `flowString`), the flow can then be executed in C++ with:

```
int result;
char * output;
EPIAVL * avl; /* populated as shown above or by code*/
if (cmd->doCommandFlow(“*”, flowString, avl,
                        &output, &result, 0, 0)
                == EPI_SUCCESS) {
    fprint(“Failed (%d)!!!\n%s\n”, result, output);
    exit(1)
}
```

Example

The following are small examples of flow code usage:

```
# determine the Passport version and save it
# for later use
@SWITCH $name d software avl
@CASE base_\([^ ,]*\)
  # $% contains the last \(\) match (the base version)
  @PRINT Passport version : $%
  @DEFINE ppversion $%
  # set variables accordingly (Tm subcomponent
  # introduced?)
  @IF $ppversion == CA.*|CB.*
    @DEFINE tm Tm
  @ELSE
    @DEFINE tm
  @END
@END
...
# use the variable defined above and set the
# p1 parameter to a default value if not set
```

```
$name set AtmIf/$atm Vcc/$vpci Vcd $tm txTdp 1 \
              ${p1:-64000}
...


# create multiple DS1 channels with @FOR
@FOR chan 0 24
  $name add Lp/$lp DS1/$ds1 Chan/$chan
  $name Lp/$lp DS1/$ds1 Chan/$chan timeslots $chan
  # run a secondary flow to create a FrUni
  # for each channel (the flow has access to
  # the current AVL including the $chan variable)
  @RUN addFrUni
@END
```

CAUTION

`DoEPITemplate`

The DoEPITemplate helps you use and invoke command flows by handling scripting aspects. You only need to create the required flow text. The utility handles everything else, including the Passport configuration pre and post-amble for the configuration flows.

See “DoEPITemplate Utility” (page 607) for the description of the DoEPITemplate utility.

Test Mode

To test command files/flows without executing them, (for example, while developing complex configuration templates) you can invoke the command with the `test` argument set to a nonzero value. This allows you to test the variable substitution, the `@SWITCH`/`@CASE` pattern matching and the general execution flow of the commands with or without actually executing them. In test mode, commands to be excuted are traced to the standard error stream (after variable substitution and prefixed by its line number). Then, a prompt usually asks for confirmation if it should be executed. The prompt depends on the command being executed:

```
@FOR <variable name> <from> <to> [<increment>]
or
@FOR <variable name> IN <token list>
```

on each iteration, the prompt offers to execute the command block or not::

> Confirm command block execution? ([y]|n|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command block is executed as normal. If n is answered, the loop's execution is terminated as though a @BREAK directive had been encountered.

@SWITCH <test command>
the prompt offers to execute the command or not:

> Execute it? (y|[n]|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered, the command is executed as normal. If n is answered (the default), you are then prompted for the output the command would have produced in order to test the @CASE pattern matching:

> Command reply? (end with @@)>

The output should be provided terminated with a line containing only the @@ characters. If you do not care to test the pattern matching, just enter @@. You will be prompted for confirmation of the match for each executed @CASE directive.

@CASE [<patterns>]
the prompt indicates if the pattern matching would succeed:

> Matches, confirm command block execution? ([y]|n|q)>

or fail:

> Does not match, execute command block anyways? (y|[n]|q)>

offering to execute the following command block or not.
If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default on success),

the command block is executed as normal. If n is answered (the default on failure) the flow is executed as though the @CASE pattern would not match and the next @CASE is block, if any, is tried instead.

@IF <variable> [==|!= <patterns>]
the prompt indicates if the test succeeds:

> Test succeeded, confirm command block execution? ([y]|n|q)>

or fails:

> Test failed, execute command block anyways? (y|[n]|q)>

offering to execute the following command block or not.
If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default on success), the command block is executed as normal. If n is answered (the default on failure) the flow is executed as though the @IF test failed and the @ELSE block, if any, is tried instead. If the command block is executed, the @ELSE block, if any, will be ignored.

@ELSE
the prompt offers to execute the following block:

> Confirm command block execution? ([y]|n|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command block is executed as normal. If n is answered, the command block is skipped until the corresponding @END construct.

@INCLUDE <command file>, or
@RUN <command file>
the prompt offers to confirm the execution of the file:

> Include/Run the file? ([y]|n|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered (the default), the command file is executed as normal. If n is answered, the command file is not included.

<device command>
the prompt offers to execute the command or not:

> Execute it? (y|[n]|q)>

If q is answered, the flow's execution is terminated as though an @EXIT directive had been encountered. If y is answered, the command is executed as normal. If n is answered (the default) the command is not executed and the next command is tried. The behavior is the same for commands prefixed with @TRACE, @NOTRACE, @TRY, and @CRITICAL.

```
@PRINT <string>
@EXIT [<code>]
@DEFINE <name> <value>
@END
```

The command is echoed as is without further prompts.

- **EPIResult** res = **NMSCmdSetFlowCBAVL**(
  EPIInterface cmd,
  char * varName,
  char * value); (C)
  EPIResult res = **NMSCmd**::**setFlowCBAVL**(
  char * varName,
  char * value); (C++)
  called within a callback (NMSCmdBindCallback()/ NMSCmd::bindCallback()) is invoked from a command flow executing (NMSCmdDoCommandFile()// NMSCmd::doCommandFile() or NMSCmdDoCommandFlow/ NMSCmd::doCommandFlow()) by the @CB construct. The callback can then inform the flow of the results by setting or resetting some variable assignments.

- **EPIResult** res = **NMSCmdOpenCommandLog(**
  EPIInterface cmd,
  char * filePath,
  int appendFlag); (C)
  EPIResult res = **NMSCmd**::**openCommandLog**(
  char * filePath,
  int appendFlag = 0); (C++)
  Opens the specified file and logs to it all commands executed with NMSCmdSendCommand()/NMSCmd::sendCommand(), and those issued by a command flow or file. Each device command executed by the flow is logged, not just the flow's execution. If appendFlag is specified as non-zero, the commands are appended to the file if it exists. By default, the file is overwritten. The log file format allows its replaying as

a command flow or as input to the `cmccmd` utility. The commands are logged from the module on, with no group name, and with all variables substituted for flow commands. Note that logging can also be controlled from the flow itself through the `@LOG` and `@LOGFILE` constructs.

- *void* **NMSCmdCloseCommandLog**(
  EPIInterface cmd); (C)
  void **NMSCmd**::**closeCommandLog**(); (C++)
  Stops command logging and closes the log file previously opened by `NMSCmdOpenCommandLog()/NMSCmd::openCommandLog()`.

Customer Database access

The Customer Database access capabilities or EPI allow you to query and modify the contents of the Customer Databases managed by Preside Multiservice Data Manager (MDM). The Customer Databases are controlled by servers (`cdbserver`) and can be accessed through the Customer Database tool (GUI). Customer Databases can also be accessed by the `cdbextract` and `cdbmerge` utilities, which are more suited for bulk extraction and population. EPI provides more programmability of the database's contents. For example, by using EPI you can receive an alarm feed from GMDR and extract the matching customer information from the database. You can then use this information to generate a trouble ticket to an external system.

Customer Database Access commands are used in the following sequence:

1 Initialize a Cdb interface.

2 Connect to the Cdb server for the appropriate database.

3 Send Fetch, Store, or Erase commands.

4 Send Query commands and receive the matching replies either synchronously with the `RecvReply` command or asynchronously through a callback function.

5 Disconnect from the Cdb server.

6 Drop the Cdb interface.

The Customer Database server is synchronous in that a Cdb interface can only perform one Fetch, Query, Store and Erase command at once. The Fetch, Store, and Erase commands are synchronous in that they always wait for the reply. The Fetch command also provides the matching information on return (similar to a non-wild card Query followed by a `RecvReply` command).

Customer Database Access commands

The following Customer Database Access commands are provided:

- **EPIInterface** cdb = **NMSCdbInit**(); (C)
  NMSCdb::**NMSCdb**(); (C++ constructor)
  Initializes a Customer Database interface.

- *void* **NMSCdbSetUserData**(
  EPIInterface api,
  void * userData); (C)
  void **NMSCdb**::**setUserData**(
  void * userData); (C++)
  void * **NMSCdbGetUserData**(
  EPIInterface api); (C)
  void * **NMSCdb**::**getUserData**(); (C++)
  void **NMSCdbSetName**(
  EPIInterface api,
  char * name); (C)
  void **NMSCdb**::**setName**(
  char * name); (C++)
  char * **NMSCdbGetName**(
  EPIInterface api); (C)
  char * **NMSCdb**::**getName**(); (C++)
  These routines allow you to associate and extract arbitrary user data and a name to the Customer Database interface.

- *void* **NMSCdbDrop**(
  EPIInterface cdb); (C)
  NMSCdb::~**NMSCdb**(); (C++ destructor)
  Drops the Customer Database interface (frees allocated resources).

- **EPIResult NMSCdbConnect**(
  EPIInterface cdb,
  char * dbName,
  char * dbHost); (C)
  EPIResult NMSCdb::**connect**(
  char * dbName,
  char * dbHost = “localhost”); (C++)

  Connects the interface to the Customer Database server for the indicated database and host.

Example (C)

```
EPIInterface cdb = NMSCdbInit();
if ( NMSCdbConnect(cdb, "EastCustDB", "bcarse88")
              == EPI_SUCCESS ) {
   ... /* we're connected */
```

Example (C++)

```
NMSCdb * cdb = new NMSCdb();
if ( cdb->connect("EastCustDB", "bcarse88")
              == EPI_SUCCESS ) {
   ... // we're connected
```

- **EPIResult NMSCdbDisconnect**(
  EPIInterface cdb); (C)
  EPIResult NMSCdb::**disconnect**(); (C++)
  Disconnects the interface from the Customer Database server. The interface may then be reconnected to another server.

- **EPIResult NMSCdbIsOK**(
  EPIInterface cdb); (C)
  int res = **NMSCdb**::**isOK**(); (C++)
  Returns current connection status of the interface (`EPI_SUCCESS`, `EPI_FAILED`, or `EPI_NO_CONNECTION`).

- **EPIResult NMSCdbFetch**(
  EPIInterface cdb,
  char * compId,
  EPITimeout timeout,
  EPICdb_Data ** data); (C)
  EPIResult NMSCdb::**fetch**(
  char * compId,
  EPICdb_Data ** data,
  EPITimeout timeout
  = **EPI_TIMEOUT_FOREVER**); (C++)
  EPIResult NMSCdbFetchHierarchical(
  EPIInterface cdb,
  char * compId,
  EPITimeout timeout,
  EPICdb_Data ** data); (C)
  EPIResult NMSCdb::**fetchHierarchical**(
  char * compId,

EPICdb_Data ** data,
EPITimeout timeout
= **EPI_TIMEOUT_FOREVER**); (C++)

Query the Customer Database for information on the specified component. The component name, `<compId>`, is typically specified in canonical format (see `NMSEPIConvertCompId` on page 349). A timeout can be specified. With a timeout of `EPI_TIMEOUT_POLL (0)`, the command acts as a no-wait poll. A value of `EPI_TIMEOUT_FOREVER (-1)` waits forever. Other values wait for the specified number of seconds.

The extracted information is returned as a pointer to a static area (if non-*`NULL`*), `<data>` upon success. This record has the following contents:

char * **compId**;
the record's component ID,

char * **relCompId**;
its related component ID,

char * **data**;
its actual data,

char * **date**;
its date, and

char * **source**;
its source code.

The returned data must not be modified nor freed. The returned pointer could also be *`NULL`*. The results of the fetch can also be extracted with the access functions: `NMSCdbGetCompID/NMSCdb::getCompID`, `NMSCdbGetRelCompID/NMSCdb::getRelCompID`,

`NMSCdbGetDate/NMSCdb::getDate`, `EPICdb_GetData/ NMSCdb::getData`, and `NMSCdbGetSource/ NMSCdb::getSource`.

If `NMSCdbFetchHierarchic/NMSCdb::fetchHierarchic` is specified, the routine will look for matching data on the parent components of the specified one if a match on it cannot be found. Also, in this mode, both the canonical and display format of the component IDs are searched for in the database.

Example (C)

```
EPICdb_Data * cdata = NULL;
if ( NMSCdbFetch(cdb, "EM TOTO FRUNI 45",
                 EPI_TIMEOUT_FOREVER, &cdata)
                 == EPI_SUCCESS ) {
   printf("Data: %s\n", cdata->data);
   ...
```

Example (C++)

```
EPICdb_Data * cdata = NULL;
if ( cdb->fetch("EM TOTO FRUNI 45", &cdata)
                 == EPI_SUCCESS ) {
   cout << "Data: " << cdata.data << endl;
   ...
```

- **EPIResult NMSCdbQuery**(
  EPIInterface cdb,
  char * compId,
  char * relCompId,
  char * data); (C)

  EPIResult NMSCdb::**query**(
  char * compId = *NULL*,
  char * relCompId = *NULL*,
  char * data = *NULL*); (C++)

  Queries the Customer Database for information matching the specified grep-style patterns (see `NMSEPIPatternMatch` on page 349). Only one pattern may be specified to match the entry's component name (`<compId>`), associated component name (`<relCompId>`), or data (`<data>`), the other arguments must be set to NULL. The command immediately returns. The replies must be extracted synchronously with

`NMSCdbRecvReply/NMSCdb::recvReply` or asynchronously through a callback function bound to the interface with `NMSCdbBindCallback/NMSCdb::bindCallback`.

Example (C)

```
if ( NMSCdbQuery(cdb, "EM TOTO FRUNI .*")
                == EPI_SUCCESS ) {
   ... /*query sent successfully */
```

Example (C++)

```
if ( cdb->query("EM TOTO FRUNI .*") == EPI_SUCCESS ) {
   ... // query sent successfully
```

- **EPIResult NMSCdbRecvReply**(
  EPIInterface cdb,
  EPITimeout timeout,
  EPICdb_Data ** data); (C)
  EPIResult NMSCdb::**recvReply**(
  EPICdb_Data ** data,
  EPITimeout timeout
  = **EPI_TIMEOUT_FOREVER**); (C++)

  Waits for and receives the next Query reply record from the Customer Database server. A timeout can be specified. With a timeout of `EPI_TIMEOUT_POLL (0)`, the command acts as a no-wait poll. A value of `EPI_TIMEOUT_FOREVER (-1)` waits forever. Other values wait for the specified number of seconds.

  `NMSCdbRecvReply` can be called for each reply, returning `EPI_SUCCESS` for each received reply. It will return `EPI_END` to indicate that no more replies are available. Further calls, as well as calls when there is no active Query command, will return an error indication.

  The resulting Customer Data entry is returned just like for `NMSCdbFetch/NMSCdb::fetch`.

  Example (C)

  ```
  EPICdb_Data * cdata = NULL;
  NMSCdbQuery(cdb, NULL, NULL, "Client Number: AP.*");
  while ( NMSCdbRecvReply(cdb, &cdata,
                  EPI_TIMEOUT_FOREVER)
                  == EPI_SUCCESS ) {
     printf("Component: %s\n", cdata->compId);
  ```

```
    printf("Updated:   %s\n", cdata->date);
    printf("Data:      %s\n", cdata->data);
    ...
```

Example (C++)

```
EPICdb_Data * cdata = NULL;
cdb->query(cdb, NULL, NULL, "Client Number: AP.*");
while ( cdb->recvReply(&cdata) == EPI_SUCCESS ) {
    cout << "Component: \n" << cdata->compId
         << "\nUpdated: \n" << cdata->date
         << "\nData:    \n" << cdata->data << endl;
    ...
```

- *char* * compId = **NMSCdbGetCompID**(
  EPIInterface cdb); (C)
  char * compId = **NMSCdb**::**getCompID**(); (C++)

  char * relcompId = **NMSCdbGetRelCompID**(
  EPIInterface cdb); (C)
  char * relcompId = **NMSCdb**::**getRelCompID**(); (C++)

  char * date = **NMSCdbGetDate**(
  EPIInterface cdb); (C)
  char * date = **NMSCdb**::**getDate**(); (C++)

  char * data = **NMSCdbGetData**(
  EPIInterface cdb); (C)
  char * data = **NMSCdb**::**getData**(); (C++)

  char * source = **NMSCdbGetSource**(
  EPIInterface cdb); (C)
  char * source = **NMSCdb**::**getSource**(); (C++)
  These functions return the contents of the last Fetch or Query reply (from `NMSCdbFetch/NMSCdb::fetch`, `NMSCdbRecvReply/NMSCdb::recvReply`, or from an asynchronous callback function bound to the Customer Database interface by `NMSCdbBindCallback/NMSCdb::BindCallback`). The returned text must not be modified nor freed. The returned pointer could also be *`NULL`*.

- **EPICdb_Data** * **NMSCdbExtractCdbData**(
  EPIInterface cdb); (C)
  EPICdb_Data * **NMSCdb**::**extractCdbData**();
  (C++)
  Extracts the Customer Data information similarly to `NMSCdbFetch/NMSCdb::fetch` from the last received reply be it from a call to `NMSCdbRecvReply/NMSAPI::recvReply` or in a bound Callback. The extracted fields, which must not be modified nor freed, are returned as a pointer to a static area valid until the next Customer Database related function call.

- **EPIResult NMSCdbBindCallback**(
  EPIInterface cdb,
  EPICallBackFunction aCbFn); (C)
  EPIResult NMSCdb::**bindCallback**(
  EPICallBackFunction aCbFn); (C++)
  Binds the specified callback function to the Customer Database interface. This callback will be executed whenever a Query command reply is received from the server for this interface. The callback is invokes as though the `NMSCdbRecvReply/NMSCdb::recvReply` function had just been called. The callback signature is:

  `typedef` *void (*`EPICallBackFunction`) (void* * cdb, *char* * text, *int* length, *int* status)*

  where `<cdb>` is an opaque pointer to the calling Customer Database interface (as returned by `NMSCdbInit/NMSCdb::NMSCdb`), `<text>` is the received matching data field, `<length>`, the data's length, and `<status>` is one of `EPI_SUCCESS`, for a successful reception, `EPI_END` to indicate an end of response, `EPI_NO_CONNECTION`, if the connection to the server was lost, and `EPI_FAILED`, for other error cases.
  The other results of the query can be extracted with the access functions: `NMSCdbGetCompID/NMSCdb::getCompID`, `NMSCdbGetRelCompID/NMSCdb::getRelCompID`, `NMSCdbGetDate/NMSCdb::getDate`, `NMSCdbGetData/NMSCdb::getData, and NMSCdbGetSource/NMSCdb::getSource`.

  Example (C)
  void
  mycb(cdb, *t*ext, length, status)
  void * cdb,

```
char * text,
int length,
int status
{
    if ( status == EPI_SUCCESS )
    {
      printf("Component: %s\n", NMSCdbGetCompID(cdb));
      printf("Updated:   %s\n", NMSCdbGetDate(cdb));
      printf("Data:      %s\n", NMSCdbGetData(cdb));
    }
    ...
}
...
NMSCdbQuery(cdb, "AP.*", NULL, NULL);
NMSCdbBindCallback(cdb, mycb);
...
NMSEPIEventLoop(); /* never returns */
```

Example (C++)

```
void
mycb(void * cdb, char * text, int length, int status)
{
    if ( status == EPI_SUCCESS )
    {
       cout << "Component: \n" << cdb->getCompID()
            << "\nUpdated: \n" << cdb->getDate()
            << "\nData:    \n" << cdb->getData()
            << endl;
    }
    ...
}
...
cdb->query("AP.*");
cdb->bindCallback(mycb);
...
NMSEPIEventLoop(); /* never returns */
```

- **EPIResult NMSCdbStore(**
  EPIInterface cdb,
  char * compId,
  char * data,
  char * relCompId,
  char * date,

char * source); (C)
EPIResult NMSCdb::**store**(
char * compId,
char * data,
char * relCompId,
char * date,
char * source = *NULL*); (C++)

Stores the specified information in the Customer Database. Typically, the component name is specified in canonical format (see `NMSEPIConvertCompId` on page 349). The information is added, or replaces that already associated with the component. The component name, `<compId>`, and data, `<data>`, must be specified. An associated component name, `<relCompId>`, a date stamp, `<date>` (defaults to the current date), and a 3-character source code, `<source>` (defaults to `"EPI"`) can also be specified or set to *`NULL`*. The command is synchronous since it blocks and waits for a confirmation from the Customer Database server.

Example (C)

```
if ( NMSCdbStore(cdb, "EM TOTO FRUNI 43",
                 "Client Number: DP542\n
Contact: (613) 763-2211)", "FRAD 213", NULL, "GCG")
                              == EPI_SUCCESS ) {
   ... /* data is now stored */
```

Example (C++)

```
if ( cdb->store("EM TOTO FRUNI 43",
                 "Client Number: DP542\n
Contact: (613) 763-2211)", "FRAD 213", NULL, "GCG")
                 == EPI_SUCCESS ) {
   ... // data is now stored
```

- **EPIResult NMSCdbErase**(
  EPIInterface cdb,
  char * compId); (C)
  EPIResult NMSCdb::**erase**(
  char * compId); (C++)

  Discards the information associated with the specified component, `<compId>`, in the Customer Database. Typically, the component name is

specified in canonical format (see `NMSEPIConvertCompId` on page 349). The command is synchronous since it blocks and waits for a confirmation from the Customer Database server.

Example (C)

```
if ( NMSCdbErase(cdb, “EM TOTO FRUNI 52”)
                == EPI_SUCCESS ) {
   ... /* data is now erased */
```

Example (C++)

```
if ( cdb->erase(“EM TOTO FRUNI 52”)
                == EPI_SUCCESS ) {
   ... // data is now erased
```

Real-Time Alarm Collection

The Real-Time Alarm Collection (RTAC) capabilities of EPI let you query the matching alarms to produce historical alarm reports. RTAC uses the rtaccol server to spool the alarms it retrieves from the GMDR server to the workstation files, one file per day (based on the alarm time stamp). With the EPI RTAC interface, you can query the spooled alarms between two date-time boundaries. You can also specify filters for any alarm attributeby using GREP-style patterns. The matching alarms are retrieved and their attributes are provided in the same way as with the Alarm&Status specialized API interface.

Use RTAC Access commands in the following sequence:

1 Initialize an RTAC interface.

2 Start a query by specifying its date/time boundaries and attribute-value filters.

3 Fetch the matching alarms.

4 For each fetched alarm, extract the desired attributes and process them.

5 Drop the RTAC interface (usually not done).

The RTAC interface is synchronous because an interface can only perform one query at a time and the Fetch command is blocking. You can, however, limit the amount of time and the number of alarms the Fetch command scans for a polling/round-robin form of multi-tasking.

You cannot use the RTAC interface remotely. You must use the RTAC interface on the same machine that stores (or NFS mounts) the RTAC spool files. The spool files are located using the `RTAC.cfg` configuration file. For details, see Real time alarm collection tool (rtaccol in 241-6001-310 *Preside MDM Server Reference Guide*.

RTAC Access commands

The following RTAC Access commands are available:

- **EPIInterface NMSRTACInit**(); (C)
  NMSRTAC::**NMSRTAC**(); (C++)
  Initializes an RTAC interface to access the RTAC database on the local workstation (as specified in the `RTAC.cfg` file).

- *void* **NMSRTACSetUserData**(
  EPIInterface rtac,
  void * userData); (C)
  void **NMSRTAC**::**setUserData**(
  void * userData); (C++)

  void * **NMSRTACGetUserData**(
  EPIInterface rtac); (C)
  void * **NMSRTAC**::**getUserData**(); (C++)

  void **NMSRTACSetName**(
  EPIInterface rtac,
  char * name); (C)
  void **NMSRTAC**::**setName**(
  char * name); (C++)

  char * **NMSRTACGetName**(
  EPIInterface api); (C)
  char * **NMSRTAC**::**getName**(); (C++)
  These routines allow you to associate and extract arbitrary user data and a name to the RTAC interface.

- *void* **NMSRTACDrop**(**EPIInterface** rtac); (C)
  virtual **NMSRTAC**::~**NMSRTAC**(); (C++)
  Destroys the specified RTAC interface (frees allocated resources).

- **EPIResult NMSRTACStartQuery**(
  EPIInterface rtac,
  char * startTime,
  char * endTime); (C)
  EPIResult NMSRTAC::**startQuery**(*char* * startTime,
  char * endTime); (C++)
  Initiates an RTAC query for the alarms within the specified start and end dates and times. Only alarms whose attributes match the specified filters (specified with `NMSRTACAddFilter`/`NMSRTAC::addFilter`) will be returned. The start time can be specified as a date ("`yyyy mm dd`"), date and time ("`yyyy mm dd hh mm ss`"), as the special values "`0000 00 00`" or "`ANY`" meaning the oldest available alarm, and as the special value "`NOW`" meaning the current (workstation) date and time. Similarly, the end date/time can be specified as a date ("`yyyy mm dd`"), date and time ("`yyyy mm dd hh mm ss`"), as the special values "`9999 99 99`" or "`ANY`" meaning the latest alarm available, or as the special value "`NOW`", meaning the current (workstation) date and time.

- **EPIResult NMSRTACAddFilter**(
  EPIInterface rtac,
  char * attName,
  char * valPattern); (C)
  EPIResult NMSRTAC::**addFilter**(
  char * attName,
  char * valPattern); (C++)
  Adds a filter to the RTAC query. The filter attribute names are the same provided through the Alarm&Status API. For details, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The filter values are specified as GREP style patterns for target values similar to the ones output by the Alarm and Status API. Multiple attribute-pattern filters can be specified by calling these functions multiple times. Filters on the same attribute are ORed together and filters on different attributes are ANDed together (similar to the API rules).

  Examples(C)

```
EPIInterface rtac = NMSRTACInit();
if (   NMSRTACStartQuery(rtac, "ANY", "NOW")
             == EPI_SUCCESS)
    && NMSRTACTAddFilter(rtac, "event" "set")
             == EPI_SUCCESS)
    && NMSRTACTAddFilter(rtac, "severity",
```

```
                "critical") == EPI_SUCCESS)
     && NMSRTACTAddFilter(rtac, "severity", "major")
                == EPI_SUCCESS) )
{
   ... /* ready to fetch all critical/major alarms up
   ...  * to now */
```

Example(C++)

```
EPIRTAC = new NMSRTAC();
char * t = NMSEPIConvertTime(EPI_SECONDS_OFFSET_TCVT,
                -7200);
if (rtac->startQuery(t, "NOW") == EPI_SUCCESS)
{
   ... // ready to fetch alarms for the last two hours
```

To initiate a new query, invoke `NMDRTACStartQuery/ NMSRTAC::startQuery` again.

- **EPIResult NMSRTACFetchNextAlarm**(
  EPIInterface rtac,
  int **maxRecords**,
  int **timeout)**; (C)
  EPIResult NMSRTAC::**fetchNextAlarm**(
  int **maxRecords** = **EPIRTAC_FOREVER**,
  int **timeout** = **EPIRTAC_FOREVER)**; (C++)
  This command fetches the next matching alarm according to the criteria specified by `NMSRTACStartQuery/NMSRTAC::startQuery and NMSRTACAddFilter/NMSRTAC::addFilter`).

  If `timeout` and/or `maxRecs` are specified as `EPIRTAC_FOREVER`, the call will block until a match is found or the end of the RTAC database is reached. Otherwise, they respectively specify the maximum number of seconds or records to try before giving up (in which case the return code is `EPI_TIMEOUT`). This allows one to "poll" the RTAC interface for a while and round-robin to other tasks.

  Example

```
...
if (   (NMSRTACStartQuery(rtac, "ANY", "NOW")
        == EPI_SUCCESS)
    && NMSRTACAddFilter("event", "set")
        == EPI_SUCCESS)
```

```
     && NMSRTACAddFilter("severity", "critical")
        == EPI_SUCCESS)
     && NMSRTACAddFilter("severity", "major")
        == EPI_SUCCESS) )
{
    while (NMSRTACFetchNextAlarm(rtac,
                EPIRTAC_FOREVER, EPIRTAC_FOREVER)
        == EPI_SUCCESS)
    {
        /* ready to extract alarm attributes */
    ...
```

The fetched alarm's contents can be extracted with the `NMSRTACExtractAlarm/NMSRTAC::extractAlarm`, `NMSRTACFormatAlarm/NMSRTAC::formatAlarm`, `NMSRTACGetFirstAttribute/NMSRTAC::getFirstAttribute`, `NMSRTACGetNextAttribute/NMSRTAC::getNextAttribute`, and `NMSRTACFindNextAttribute/NMSRTAC::findNextAttribute` routines.

- **EPIGMDRAPI_Alarm * NMSRTACExtractAlarm(**
  EPIInterface api); (C)
  EPIGMDRAPI_Alarm * NMSRTAC::**extractAlarm**(); (C++)
  Extracts the major attributes of the last alarm fetched by `NMSRTACFetchNextAlarm/NMSRTAC::fetchNextAlarm` (similarly to `EPIGMDRAPIExtractAlarm/EPIGMDRAPI::extractAlarm`). The extracted fields are returned into a static area and a pointer to that area is returned as the `<alarm>` argument. This static area of type `EPIGMDRAPI_Alarm` contains the following fields whose values must not be modified nor freed (they are valid until the next RTAC related function call):

  *char * **compId**;*
  is the alarm's component ID..

  *char * **time**;*
  is the alarm's time stamp in API format.

  *char * **severity**;*
  is the alarm's severity value.

char * **event**;
is the alarm's event type.

char * **faultCode**;
is the alarm's fault code.

char * **operatorData**;
is the alarm's operator data.

char * **rawState**;
is the alarm's raw state.

Note: Some members may be *NULL* if the corresponding field is not present. As well, `operatorData` can consist of more than one line.

For information on the legal values of these fields, see 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*. The other attributes, if any were specified, can be extracted with the `NMSRTACGetFirstAttribute/NMSRTAC::getFirstAttribute`, `NMSRTACGetNextAttribute/NMSRTAC::getNextAttribute`, and `NMSRTACFindNextAttribute/NMSRTAC::findNextAttribute` commands.

Example (C)

```
...
if ( NMSRTACRecvNextAlarm(rtac,
                EPIRTAC_FOREVER, EPIRTAC_FOREVER)
                == EPI_SUCCESS ) {
    EPIGMDRAPI_Alarm * alarm;
    if (   (alarm = NMSRTACExtractAlarm(rtac)) {
        && (strcasecmp(alarm->severity,
                "critical") == 0) ) {
                ... /* handle the critical alarm */
```

Example (C++)

```
...
if ( rtac->recvAlarm(alarm) == EPI_SUCCESS ) {
    EPIGMDRAPI_Alarm * alarm;
    if (   (alarm = rtac->extractAlarm())
```

```
        && (strcasecmp(alarm->severity,
              "critical") == 0) {
              ... // handle the critical alarm
```

- *void* **NMSRTACResetAttributes**(
  EPIInterface api); (C)
  void **NMSAPI**::**resetAttributes**(); (C++)

  EPIRTACAV * **EPIRTAC_GetFirstAttribute**(
  EPIInterface rtac); (C)
  EPIRTACAV * **EPIRTAC**::**getFirstAttribute**(); (C++)

  EPIRTACAV * **EPIRTAC_GetNextAttribute**(
  EPIInterface rtac); (C)
  EPIRTACAV * **EPIRTAC**::**getNextAttribute**(); (C++)

  EPIRTACAV * **EPIRTAC_FindNextAttribute**(
  EPIInterface rtac,
  char * attrName); (C)
  EPIRTACAV * **EPIRTAC**::**findNextAttribute**(
  char * attrName); (C++)
  These routines allow you to scan the list of attributes of the last fetched alarm. `EPIRTACResetAttributes/EPIRTAC::resetAttributes` resets the scanning to the beginning of the list. `EPIRTACGetFirstAttribute/EPIRTAC::getFirstAttribute` extracts the first attribute in the list. `EPIRTACGetnextAttribute/EPIRTAC::getNextAttribute` returns the next attribute in the list. `EPIRTACFindnextAttribute/EPIRTAC::findNextAttribute` returns the next attribute in the list whose attribute name (NRS type name or long name/title) matches the specified one. The attributes are returned as pointers to a strucutre of type `EPIRTACAV` which contains the following fields:

  char * **name**
  is the attribute name.

  char * **value**

is the attribute value.

The pointer and its contents must not be modified or deleted.

Example (C)

```
...
EPIRTACAV * att;
while ( ( att = NMSRTACFindNextAttribute(rtac,
              "time") ) != NULL ) {
   printf("Time is: %s\n", att->value);
...
```

Example (C++)

```
...
EPIRTACAV * att;
while ( ( att = rtac->findNextAttribute("time") )
              != NULL ) {
   cout << "Time is: " << att->value << endl;
...
```

- *char* * **NMSRTACFormatAlarm**(
  EPIInterface rtac,
  EPIRTAC_ALARM_FORMATS format); (C)
  char * **NMSRTAC**::**formatAlarm**(
  EPIRTAC_ALARM_FORMATS format); (C++)
  This command produces the last received alarm (from `NMSRTACRecvNextAlarm/NMSRTAC::recvNextAlarm`) in the Preside Multiservice Data Manager (MDM) Common Alarm Format (as used in the Alarm Display and Component Information Viewer tools). The alarm can be formatted `<format>`, using the following

EPI_ALARM_TERSE_FORMAT	one line summary
EPI_ALARM_NORMAL_FORMAT	includes Comment Data
EPI_ALARM_FULL_FORMAT	all information is displayed

  The default is `EPI_ALARM_FULL_FORMAT`. The returned text buffer contains the formatted alarm. Unlike other EPI strings, you need to ensure that your program releases this buffer when you are done with it.

Example

```
...
    while (NMSRTACFetchNextAlarm(rtac) == EPI_SUCCESS)
    {
        EPIRTACAV * att = NMSRTACFindNextAttribute(
                rtac, “compCriticality”);
        if (att && (atoi(att->value) > 60))
        {
                char * str = NMSRTACFormatAlarm(rtac,
                        EPI_ALARM_FULL_FORMAT);
                printf(“%s\n”, str);
        }
    }
```

Network Reporting System (NRS)

The Network Reporting System (NRS) capabilities of EPI let you create device configuration reports on the data stored in the NRS database and its schema (RDF files). You build the NRS query by identifying the node configuration files to scan (by name, pattern, and naming discipline such as keyed and dated)and the component types to extract configuration data from. The configuration data is returned to the EPI script in various ways (for example, environment variable, associative arrays, and standard output). The data values correspond to the way NRS reports currently work. The EPI NRS capabilities also add value by providing program access to the following items:

- NRS schema contents (the RDF files)
- the automatic construction of the component name
- the ability to specify Passport component and attribute types by name and heirarchical path, rather than by numerical IDs.

EPI does not provide a means of populating this database. You must use the NRS utilities for this (some examples include nrspop, pnrspop, and sisauto). For more information on the NRS database and its use, see 241-6001-022 *Preside MDM Network Reporting System User Guide.*

Use the following sequence for NRS Access commands:

1 Initialize an NRS interface.

2 Start a query by specifying the target node configurations and the component types to be reported, and/or

3 Load and interrogate the NRS schema to help create the report (optional).

4 Fetch the matching configuration components. Components are extracted in depth-first order with no garanteed ordering at peer level (as with other NRS reporting mechanisms).

5 For each fetched component, extract the desired attributes and process them.

6 Drop the NRS interface (usually not done).

The NRS interface is synchronous because an interface can only perform one query at a time and the Fetch command is blocking. You can, however, limit the amount of time and the number of alarms the Fetch command scans to support a polling/round-robin form of multi-tasking.

You cannot use the NRS interface remotely. Use the NRS interface on the same machine that stores (or NFS mounts) the NRS database. The database and schema are located using the `NRS.cfg` configuration file. For details, see 241-6001-022 *Preside MDM Network Reporting System User Guide*.

The EPI NRS interface is one of the most involved due to the complexity of the data structure being manipulated. The data elements manipulated by the interface (and to which it provides access routines) are shown in the figures “EPI NRS data element and input operations” (page 454) and “EPI NRS data element and output operations” (page 455).

Figure 1
EPI NRS data element and input operations

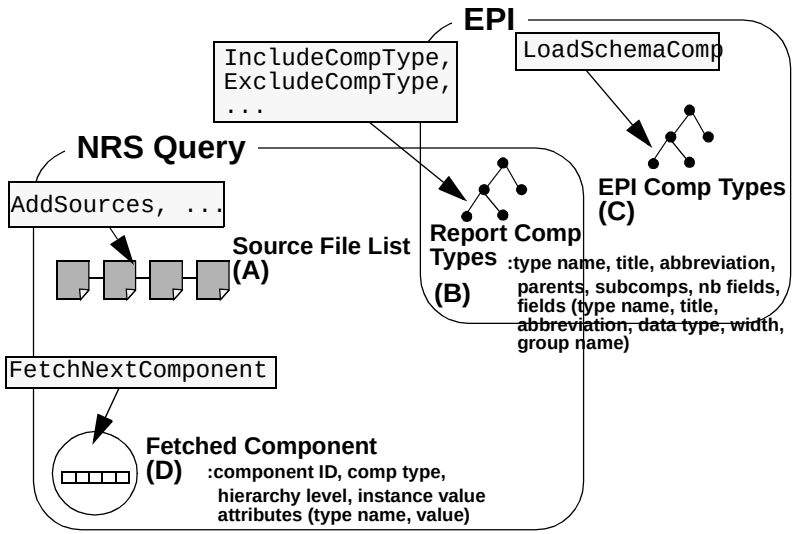
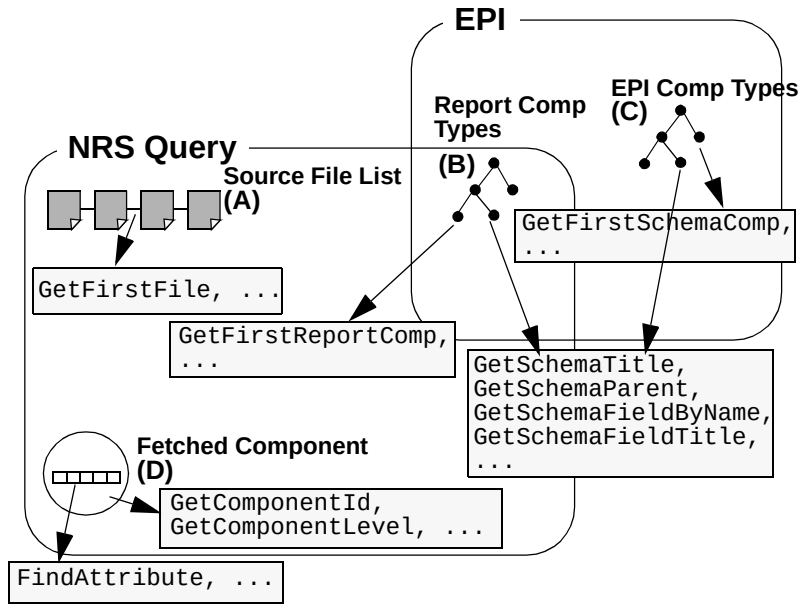


Figure 2
EPI NRS data element and output operations



Legend:

— (A) is a list of NRS data files to be scanned and reported on (constructed with `NMSNRSAddSources/NMSNRS::addSources` and other related calls, scanned with `NMSNRSGetFirstFile/ NMSNRS::getFirstFile` and related calls),

— (B) is list of component types to be extracted by the query (constructed with `NMSNRSIncludeCompType/ NMSNRS::includeCompType` and `NMSNRSExcludeCompType/ NMSNRS::excludeCompType`, scanned with `NMSNRSGetFirstReportComp/ NMSNRS::getFirstReportComp` and related calls). These component types form a partial hierarchy that can be examined with a number of routines to extract their parameters (for example, `NMSNRSGetSchemaTitle/NMSNRS::getSchemaTitle`), topology (for example, `NMSNRSGetFirstSchemaSubcomp/`

`NMSNRS::getFirstSchemaSubcomp` and `NMSNRSGetFirstSchemaParent/ NMSNRS::getFirtsSchemaParent`), and component attributes (for example, `NMSNRSGetSchemaFieldByName/ NMSNRS::getSchemaFieldByName`, `NMSNRSGetSchemaFieldTitle/ NMSNRS::getSchemaFieldTitle`). Schema components are manipulated in the interface through an opaque pointer.

— (C) is a list of all component types (RDFs) known to the EPI interface -- a superset of the list of component types to be reported on just mentioned -- (for example, accumulated with the include/ exclude calls above and the `NMSNRSLoadSchemaComp/ NMSNRS::loadSchemaComp` routine, examined by `NMSNRSGetFirstSchemaComp/ NMSNRS::getFirstSchemaComp` and related calls). This list is manipulated directly only if you want to examine the NRS RDF Schema independently of a query.

— (D) is the last component fetched by the query including its parameteres (for example, `NMSNRSGetComponentId::NMSNRS::getComponentId`, and `NMSNRSGetComponentLevel::NMSNRS::getComponentLevel`) and its attributes (for example, `NMSNRSFindAttribute/ NMSNRS::findAttribute`). If a NULL value is passed to the Schema access routines above, the component type of the last fetched component is used.

NRS Access commands

- **EPIInterface NMSNRSInit**(); (C)
  NMSNRS::**NMSNRS**(); (C++)
  Initializes an NRS interface.

- *void* **NMSNRSSetUserData**(
  EPIInterface nrs,
  void * userData); (C)
  void **NMSNRS**::**setUserData**(
  void * userData); (C++)
  void * **NMSNRSGetUserData**(
  EPIInterface nrs); (C)
  void * **NMSNRS**::**getUserData**(); (C++)

void **NMSNRSSetName**(
EPIInterface nrs,
char * name); (C)
void **NMSNRS**::**setName**(
char * name); (C++)

char * **NMSNRSGetName**(
EPIInterface nrs); (C)
char * **NMSNRS**::**getName**(); (C++)
These routines allow you to associate and extract arbitrary user data and a name to the NRS interface.

- *void* **NMSNRSDrop**(**EPIInterface** nrs); (C)
virtual **NMSNRS**::~**NMSNRS**(); (C++)
Drops the NRS interface (frees allocated resources).

- **EPIResult NMSNRSStartQuery**(
EPIInterface nrs,
char * deviceType,
char * nodePattern,
char * versionPattern); (C)
EPIResult NMSNRS::startQuery(
char * deviceType = *NULL*,
char * nodePattern = *NULL*,
char * versionPattern = *NULL*); (C++)
Initiates an NRS query. If non-NULL, the arguments specify the NRS data files to be scanned by the query (see `NMSNRSAddSources/NMSNRS::addSources` below for an explanation of these arguments). Additional data files can be specified using the `NMSNRSAddSources/NMSNRS::addSources`, `NMSNRSAddSource/NMSNRS::addSource`, `NMSNRSAddKeyedSources/NMSNRS::addKeyedSources`, `NMSNRSAddDatedSources/NMSNRS::addDatedSources`, and/or `NMSNRSAddLatestSources/NMSNRS::addLatestSources` routines. The component types to report from the files are specified using the `NMSNRSIncludeCompType/NMSNRS::includeCompType` and `NMSNRSExcludeCompType/NMSNRS::excludeCompType` routines.

  To start a new NRS query, call `NMSNRSStartQuery/NMSNRS::startQuery` again.

- **EPIResult NMSNRSAddSources**(
  EPIInterface nrs,
  char * deviceType,
  char * nodePattern,
  char * versionPattern); (C)
  EPIResult NMSNRS::**addSources**(
  char * deviceType,
  char * nodePattern,
  char * versionPattern); (C++)

  EPIResult NMSNRSAddSource(
  EPIInterface nrs,
  char * fileName); (C)
  EPIResult NMSNRS::**addSource**(
  char * fileName); (C++)

  EPIResult NMSNRSAddKeyedSources(
  EPIInterface nrs,
  char * deviceType,
  char * nodePattern,
  char * keyPattern); (C)
  EPIResult NMSNRS::**addKeyedSources**(
  char * deviceType,
  char * nodePattern,
  char * keyPattern); (C++)

  EPIResult NMSNRSAddDatedSources(
  EPIInterface nrs,
  char * deviceType,
  char * nodePattern,
  char * datePattern); (C)
  EPIResult NMSNRS::**addDatedSources**(
  char * deviceType,
  char * nodePattern,
  char * datePattern); (C++)

  EPIResult NMSNRSAddLatestSources(
  EPIInterface nrs,
  char * deviceType,

char * nodePattern,
char * fromDateTime); (C)

EPIResult NMSNRS::**addLatestSources**(
char * deviceType,
char * nodePattern,
char * fromDateTime = *NULL*); (C++)

These routines specify which NRS data files to report on. The files are located at the path specified in the `NRS.cfg` file. To details about populating the NRS database, see 241-6001-022 *Preside MDM Network Reporting System User Guide*. The file(s) can be specified in a number of ways;

`NMSNRSAddSources/` `NMSNRS::addSources`	Includes NRS data file(s) matching the specified GREP patterns; `<deviceType>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<nodePattern>`, the module name, and `<versionPattern>`, the configuration file name.
`NMSNRSAddSource/` `NMSNRS::addSource`	Explicitly names the NRS data file to report on (with or without a full path -- the NRS database path as configured in `NRS.cfg` will be used if not specified).

`NMSNRSAddKeyedSources/ NMSNRS::addKeyedSources`	Includes NRS data file(s) matching the specified GREP patterns; `<devicePattern>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<nodePattern>`, the module name, and `<keyPattern>`, the configuration file key. Keyed configuration file names have a fixed prefix (the key) followed by a variable two-digit counter. `<keyPattern>` only matches the key prefix. For a matching key, the file with the highest two-digit suffix is selected.
`NMSNRSAddDatedSources/ NMSNRS::addDatedSources`	Includes NRS data file(s) matching the specified GREP patterns; `<devicePattern>`, the device type (for example, `ppc`, `ppe`, `dpn`), `<nodePattern>`, the module name, and `<datePattern>`, the configuration file date (six digits, not a pattern). Dated configuration file names have a fixed six-digit prefix (the date) followed by a variable two-digit counter. `<datePattern>` only matches the date prefix. For Passports, `<datePattern>` matches the activation date of the NRS data file name. For both Passport and DPN files, the highest dated file up to the specified date is accepted (and the one with the highest two-digit counter suffix).

`NMSNRSAddLatestSources/ NMSNRS::addLatestSources`	Includes NRS data file(s) matching the specified GREP patterns; `<devicePattern>`, the device type (for example, `ppc`, `ppe`, and `dpn`), and `<nodePattern>`, the module name. If you specify `<fromDateTime>` as `"YYMMDD"`, `"YYYY MM DD"`, or `"YYYY MM DD HH MM SS"`, only the NRS data files from that date/time forward (UNIX file modifitation time-stamp) are considered. This option is useful for creating incremental reports based on the latest NRS population or the last report invocation. For each matching module, the matching file with the highest file system date is selected (the most recently populated file).

In all cases where multiple files match the patterns provided in one of these calls, only one file per module is selected, that is, the highest one in alphanumerical order. For Passport, this also means the one with the highest version counter, the three -igit file suffix). Multiple calls to these routines can select multiple files for the same module.

Example(1) (C)

This example and the following assume a sample NRS database containing the following files:

```
dpn.R78.4078.R7872.970709.data
dpn.R78.4078.R7888.970715.data
ppc.NODER16.2105.NEWCARD,full,012.001105.data
ppc.NODER16.2105.NEWCARD05,full,001.001102.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
ppc.NODER17.2106.newconf,full,030.010210.data
ppc.NODER17.2106.demo,full,003.010123.data
ppc.EASTOTT.2100.lab,full,005.010123.data
ppc.EASTOTT.2100.lab,full,011.010124.data
```

```
    ppc.EASTMTL.2109.demo,full,031.010104.data
    ppe.SOUTHRICH.2102.JF,full,003.010124.data
    ppe.SOUTHRTP.2113.voice,full,044.010124.data

EPIInterface * nrs = NMSNRSInit();
if (   (NMSNRSStartQuery(nrs) == EPI_SUCCESS)
     && (NMSNRSAddSource(nrs,
           "dpn.R78.4078.R7872.970709.data")
           == EPI_SUCCESS) )
{
    ... /* ready to fetch components from
    ...  *the specified NRS data file */
```

Given the preceding sample database, this call selects the following file:

```
    dpn.R78.4078.R7872.970709.data
```

Example(2) (C++)

```
NMSNRS nrs();
if (   (nrs.NMSNRSStartQuery() == EPI_SUCCESS)
     && (nrs.NMSNRSAddSources("ppc", ".*",
           "NEWCARD.*")
           == EPI_SUCCESS) )
{
    ... // ready to fetch components from all
    ... // node configurations whose name start with
    ... // NEWCARD
```

Given the preceding sample database, this call selects the following files:

```
    ppc.NODER16.2105.NEWCARD,full,012.001105.data
```

(the "latest" of the two matching files for NODER16)

```
    ppc.NODER12.2101.NEWCARD09,full,001.001102.data
```

Example(3) (C)

```
EPIInterface * nrs = NMSNRSInit();
if (   (NMSNRSStartQuery(nrs) == EPI_SUCCESS)
     && (NMSNRSAddKeyedSources(nrs, "ppc", ".*",
           "DEMO")
           == EPI_SUCCESS) )
{
    ... /* as above but this time for keyed
    ...  * configurations with DEMO as the key */
```

Given the sample database, this call selects the following files:

```
    ppc.NODER16.2105.NEWCARD05,full,001.001102.data
```

```
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
```

(The other NODER16 file was not selected as its name does not match the syntax of a keyed file)

Example(4) (C++)

```
NMSNRS nrs();
if (   (nrs.NMSNRSStartQuery() == EPI_SUCCESS)
    && (nrs.NMSNRSAddDatedSources("ppc", ".*",
          "010211") == EPI_SUCCESS) )
{
   ... // as above but this time for all node
   ... // configurations dated before or for February
   ... # 11th 2001
```

Given the sample database, this call selects the following files:

```
ppc.NODER16.2105.NEWCARD,full,012.001105.data
ppc.NODER12.2101.NEWCARD09,full,001.001102.data
ppc.NODER17.2106.newconf,full,030.010210.data
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
```

(All configurations up to 010211 for PPC nodes are taken.)

Example(5) (C)

```
EPIInterface * nrs = NMSNRSInit();
if (   (NMSNRSStartQuery(nrs) == EPI_SUCCESS)
    && (NMSNRSAddLatestSources(nrs, "ppc", "EAST.*")
          == EPI_SUCCESS)
    && (NMSNRSAddLatestSources(nrs, "ppe", "SOUTH.*")
          == EPI_SUCCESS) )
{
   ... /* ready to fetch Software components from the
   ...  * lastest configuration of the nodes whose
   ...  * names start with EAST or SOUTH */
```

Given the sample database, this call selects the following files:

```
ppc.EASTOTT.2100.lab,full,011.010124.data
ppc.EASTMTL.2109.demo,full,031.010104.data
ppe.SOUTHRICH.2102.JF,full,003.010124.data
ppe.SOUTHRTP.2113.voice,full,044.010124.data
```

(Assuming the UNIX file data matches the ordering of the activation dates in the file names.)

- *void* **NMSNRSResetFiles**(
  EPIInterface nrs,); (C)
  void **NMSNRS**:::**resetFiles**(); (C++)

  char * **NMSNRSGetFirstFile**(
  EPIInterface nrs); (C)
  char * **NMSNRS**::**getFirstFile**(); (C++)

  char * **NMSNRSGetNextFile**(
  EPIInterface nrs); (C)
  char * **NMSNRS**::**getNextFile**(); (C++)
  These routines provide access to the list of NRS data files to be scanned by the query (as inserted by `NMSNRSAddSources/ NMSNRS::addSources` and its related calls).

  Example(C++)

```
NMSNRS nrs();
if (   (nrs.NMSNRSStartQuery() == EPI_SUCCESS)
    && (nrs.NMSNRSAddDatedSources("ppc", ".*",
          "000711") == EPI_SUCCESS) )
{
   printf("Reporting on files: \n");
   for (char * fname = nrs.getFirstFile();
        fname;
        fname = nrs.getNextFile() )
          printf(" %s\n", fname);
   ...
```

- **EPIResult NMSNRSIncludeCompType**(
  EPIInterface nrs,
  char * compType,
  int withNext,
  int withSubcomps,
  int withParents); (C)
  EPIResult NMSNRS::**includeCompType**(
  char * compType,
  int withNext = 0,
  int withSubcomps = 1,
  int withParents = 1); (C++)

  EPIResult NMSNRSExcludeCompType(

EPIInterface nrs,
char * compType,
int withNext,
int withSubcomps,
int withParents); (C)

EPIResult NMSNRS::**excludeCompType**(
char * compType,
int withNext = 0,
int withSubcomps = 1,
int withParents = 1); (C++)

- These routines add or remove component types to the list of component types to be reported by the query. By default, no types are reported. Component types are specified as strings with the format

  `[<device type>/]<component>`

  where `<device type>` is a recognized NRS device type (`ppc` for Carrier versions of Passports, `ppe` for Enterprise versions, `dpn` for DPN equipment). If no device type is specified, the default type for the installation is taken from the `NRS.cfg` file. The `<component>` specification can be a name for DPN components, a numerical component ID for Passport components, or a name (the full component type name, for example, `FrameRelayUni`, or the prompt, for example, `FrUni`) for Passport components.

  Note: The Passport component names are not unique since there can be multiple Passport components with the same name or prompt. Passport component types are uniquely identified by their numeric component ID. Including or excluding component types to the report by name includes or excludes all the possible matches. This may result in unwanted components being reported.

  Passport components can also be identified as the full hierarchy of full names and/or abbreviations. For example, to include the `ServiceParameter` component of a Frame Relay UNI Dlci, the following component type can be specified: `ppc/EM-FrUni-Dlci-Sp`. In that case, only the specified Sp subcomponent (FrUni-Dlci-Sp) will be returned. `Sp` components of other hierarchies (for example, FrNni-Dlci-Sp) will be ignored.

The names are not case sensitive. The schema, RDF files, to interpret the names are located as specified in the `NRS.cfg` file.

If `<withSubcomps>` is given a nonzero value, all the component's possible subcomponents (recursively) are also included/excluded. If `<withNext>` is given a nonzero value, its immediate subcomponents are also included/excluded. If `<withParents>` is given a nonzero value, then all the component's possible parent components are also included/excluded. If the modifier is applied to a full path Passport component specification, then only the related components of the specified full path are added. In non-path specifications, all possible related components are included or excluded. It is possible to include whole sub-hierarchies of components then exclude the subcomponents that are not needed by calling these routines repreatedly. It is also possible to change the list of component types to report on the fly as the components are reported.

Example(1) (C)

```
...
if (   (NMSNRSIncludeCompType(nrs, "ppc/2", 0, 0, 0)
          == EPI_SUCCESS)
    && (NMSNRSIncludeCompType(nrs,
          "ppc/FrameRelayUni", 0, 1, 0)
          == EPI_SUCCESS) )
{
   ... /* ready to specify the source files for an NRS
   ...  * report on the (Carrier) Passport module and
   ...  * Frame Relay components (including all its
   ...  * subcomponents) */
```

Example(2) (C)

```
if (   (NMSNRSIncludeCompType(nrs, "ppc/2", 0, 0, 0)
          == EPI_SUCCESS)
    && (NMSNRSIncludeCompType(nrs,
          "ppc/FrameRelayUni", 0, 1, 0)
          == EPI_SUCCESS)
    && (NMSNRSExcludeCompType(nrs, "ppc/Signalling",
          0, 0, 0)
          == EPI_SUCCESS)
    && (NMSNRSExcludeCompType(nrs,
          "ppc/DataLinkConnectionIdentifier", 0, 1, 0)
          == EPI_SUCCESS) )
{
```

```
... /* Same as previous example except that this
...  * time the Signalling and DLCI subcomponents
...  * (and all its subcomponents for the later) are
...  * to be excluded from the report) */
```

Example(3) (C++)

```
...
if (nrs->includeCompType("ppc/EM-AtmIf-Vcc-Vcd"
          0, 0, 1)
          == EPI_SUCCESS)
{
   ... // ready to specify the source files for an NRS
   ... // report on the (Carrier) Atm Virtual
   ... // Connection Description and its indicated
   ... // parents (Vcc, AtmIf and EM).
```

- *void* **NMSNRSResetReportComps**(
  EPIInterface nrs,); (C)
  void **NMSNRS**:::**resetReportComps**(); (C++)
  EPINRSSchemaComp NMSNRSGetFirstReportComp(
  EPIInterface nrs); (C)
  EPINRSSchemaComp NMSNRS::**getFirstReportComp**(); (C++)
  EPINRSSchemaComp NMSNRSGetNextReportComp(
  EPIInterface nrs); (C)
  EPINRSSchemaComp NMSNRS::**getNextReportComp**(); (C++)
  These routines provide access to the list of component types to be extracted by the query. The returned value is an opaque pointer that can be used in the routines described below (`NMSNRSGetSchemaName/ NMSNRS::getSchemaName`, ...) to extract the schema component's parameters.

  Example(C++)

```
if ((NMSNRSIncludeCompType(nrs,
          "ppc/FrameRelayUni", 0, 1, 0)
          == EPI_SUCCESS) )
{
   printf("Reporting on components: \n");
   for (EPINRSSchemaComp comp
          = nrs.getFirstReportComp();
        comp;
        comp = nrs.getNextReportComp() )
          printf(" %s\n", nrs.getSchemaTitle(comp));
   ...
```

- **EPIResult NMSNRSFetchNextComponent(**
  EPInterface nrs,
  int skipToLevel,
  int stopAtLevel,
  int timeout,
  int maxRecs); (C)

  EPIResult NMSNRS::**fetchNextComponent(**
  int skipToLevel = **EPI_NO_SKIP**,
  int stopAtLevel = **EPI_NO_SKIP**,
  int timeout = **EPI_FOREVER**,
  int maxRecs = **EPI_FOREVER**); (C++)

  Fetches the next matching component (according to the component types specified in `NMSNRSIncludeCompType/ NMSNRS::includeCompType and NMSNRSExcludeCompType/ NMSNRS::excludeCompType`) from the selected (through `NMSNRSAddSources/NMSNRS::addSources and other related calls`) NRS data files.

  The fetched components consist of a number of contextual information (which include component name, NRS type name, instance value, hierarchy level, and the name of the NRS data file it was extracted from) plus a list of attributes (attribute value pairs) specific to the device vintage (RDF). This information, as well as the schema information for the current component, is available through a series of routines described below.

  If the `<skipToLevel>` is set to a value different than `EPI_NO_SKIP`, the command returns the next matching component of a hierachy level that is smaller or equal to the one given. Other matching components of a higher level are ignored. This restricts the search by skipping components that are undesirable (for example. if a Service Parameter (level 3) component attribute of a Passport Frame Relay DLCI component (level 2) does not meet the necessary criteria, the next call to `NMSCmdFetchNextComponent/NMSNRS::fetchNextComponent` can specify a skip level of `2` which skips to the next DLCI component, ignoring any intervening matches). NRS components are delivered in depth-first order.

Note: Specifying a skip level of `0` is an efficient way of skipping an entire module (configuration file) since EPI does not try to match the remaining components. The command tries to match components in the next module-configuration specified with `NMSNRSAddSources/ NMSNRS::addSource` (and related calls). Specifying a skip level of `-l` has the same effect as not specifying the option at all.

If you use the `<stopAtLevel>` option, the next matching component is returned as normal. In addition, if a non-matching component of the specified level or a lower value is found, the command returns with an empty component All variables are `NULL` except for `the hierarchy level`). As a result, you are informed when the member of a component sub-tree is scanned and another sub-tree of the same level is about to be scanned (the new sub-tree matches, then it is returned as normal, else the empty component is returned as a form of component terminator). For example, assuming `FrUnis` (level 1) are being extracted with a stop level of `1` specified as soon as the first one is found, all matching `FrUnis` will be returned as normal and an empty component is returned once a `Vs` components (also at level 1 but not requested by the query) is found. This technique allows one to maintain a state machine that knows when not to expect more sub-components of a specific sub-tree.

If you set the `<timeout>` and/or `<maxRecs>` options to a value other than `EPI_FOREVER`, the search does not continue beyond the specified timeout (in seconds) and maximum number of scanned records respectively. If a timeout or maximum number of records tested occurs, the command returns the exit code `EPI_TIMEOUT`. This let you “poll” the NRS interface for a while and round-robin to other tasks.

Example(1) (C)

```
/* looking for FrUnis with no running LMIs */
...
NMSNRSIncludeCompType(nrs, “ppc/FrUni”, 1, 0, 0);
...
int skip = EPI_NO_SKIP;
while (NMSNRSFetchNextComponent(nrs, skip,
          EPI_FOREVER, EPI_FOREVER) == EPI_SUCCESS)
{
   skip = EPI_NO_SKIP;
   EPINRSAV * av = NMSNRSFindAttribute(nrs,
```

```
            "procedures");
    if (   (strcmp(NMSNRSGetComponentAbbref(nrs),
                        "Lmi") == 0)
        && av && (strcmp(av->value, "none") == 0))
    {
            printf("%s has no Lmi",
                        NMSNRSGetComponentId(nrs));
    } else {
            /* skip to the next FrUni */
            skip = NMSNRSGetComponentLevel(nrs) - 1;
    }
}
```

- *char* * **NMSNRSGetComponentMarker**(
  EPInterface nrs); (C)
  char * **NMSNRS::getComponentMarker**(); (C++)

  EPIResult NMSNRSSeekComponentMarker(
  EPInterface nrs,
  char * seekMarker); (C)
  EPIResult NMSNRS::seekComponentMarker(
  char * seekMarker); (C++)
  `EPINRSGetComponentMarker`/`EPINRS::getComponentMarker` returns a marker for the current fetched component (or `NULL` on failure). You can save the marker and use it later as an argument to `NMSNRSSeekComponentMarker/NMSNRS::seekComponentMarker` to relocate a component structure which is helpful when the NRS peer component type order is not specified. All instances of a subcomponent type X are together, but it is not specified if instances of type X appear before or after those of its peer type Y. For example, an NRS interface may be used to scan for Frame Relay interfaces and their DNA subcomponents. When an FrUni is found, its marker is saved and the interface is used to locate its Dna. The same or another NRS interface can be used to scan for the FrUni's Dlci subcomponents. The NRS interface requires one to manipulate the selected types with `NMSNRSIncludeCompType/NMSNRS::includeCompType and NMSNRSExcludeCompType/NMSRNS::excludeCompType`. The marker may be exchanged between properly initialized NRS interfaces. Markers may also refer to different files. If the marker cannot be computed, an empty string is returned.

Example(C++)

```
// Assume two NRS interfaces (FrUniNRS and DlciNRS)
// configured to scan for FrUnis/Dnas and Dlcis
// respectively
char marker[128];
while ( FrUniNRS->fetchNextComponent() {
    if ( strcasecmp(FrUniNRS->getComponentAbbrev(),
                    "FRUNI") == 0 ) {
          // ... process FrUni information...
          strcpy(marker,
             FrUniNRS->getComponentMarker());
    } else if ( strcasecmp(
                   FrUniNRS->getComponentAbbrev(),
                   "DNA") == 0 ) {
          // ... process DNA information ...
          DlciNRS->seekComponentMarker(marker);
          int stopAt = -1;
          while ( DlciNRS->fetchNextComponent( \
                          EPI_NO_SKIP, stopAt)
                       == EPI_SUCCESS ) {
             stopAt = 1;
             //... process the DLCIs ...
          }
    }
}
```

- *char* * **NMSNRSGetComponentId**(
  EPIInterface nrs); (C)
  char * **NMSNRS**::**getComponentId**(); (C++)

  char * **NMSNRSGetComponentType**(
  EPIInterface nrs); (C)
  char * **NMSNRS**::**getComponentType**(); (C++)

  char * **NMSNRSGetComponentAbbrev**(
  EPIInterface nrs); (C)
  char * **NMSNRS**::**getComponentAbbrev**(); (C++)

  char * **NMSNRSGetComponentTitle**(
  EPIInterface nrs); (C)
  char * **NMSNRS**::**getComponentTitle**(); (C++)

char * **NMSNRSGetComponentValue**(
EPIInterface nrs); (C)
char * **NMSNRS**::**getComponentValue**(); (C++)

int **NMSNRSGetComponentLevel**(
EPIInterface nrs); (C)
int **NMSNRS**::**getComponentLevel**(); (C++)

char * **NMSNRSGetCurrentFilePath**(
EPIInterface nrs); (C)
char * **NMSNRS**::**getCurrentFilePath**(); (C++)

These routines extract the general information on the last matching component fetched.

`NMSNRSGetComponentId/NMSNRS::getComponentId` returns the full Component ID of the fetch component (the component is specified in mixed case, with space separators, for example, "`EM NODER16 FrUni 132 Dlci 206 Sp $`", and can be manipulated with `NMSEPIConvertCompId`).

`NMSNRSGetComponentType/NMSNRS::getComponentType` returns the component type of the fetched component (the component type is specified as `<device>/<type>` where `<device>` is the NRS device type (`ppc`, `ppe`, or `dpn`), and `<type>` is the component type (a name for DPN, a numerical component ID for Passport, for example, `dpn/PE` or `ppe/8664`).

`NMSNRSGetComponentAbbrev/NMSNRS::getComponentAbbrev` returns the component short type name (abbreviation).

`NMSNRSGetComponentTitle/NMSNRS::getComponentTItle` returns the component long type name.

`NMSNRSGetComponentValue/NMSNRS::getComponentValue` returns instance value of the fetched component (the value of its most specific component ID category/value pair, for example, "`$`" for the component identified above).

`NMSNRSGetComponentLevel/NMSNRS::getComponentLevel` returns the hierarhcy level of fetched component (starting at 0 for the module level, for example, `3` for the component identified above). Note that matching components are returned in depth-first order.

`NMSNRSGetCurrentFilePath/NMSNRS::getCurrentFilePath` returns the NRS data file from which the matching component was fetched.

Example
The following table shows the possible results from a fetched component:

Command	Returned value
`NMSNRSGetComponentId/ NMSNRS::getComponentId`	`"EM NODER16 FrUni 132 Dlci 206 Sp $"`
`NMSNRSGetComponentType/ NMSNRS::getComponentType`	`"ppe/8664"`
`NMSNRSGetComponentAbbrev/ NMSNRS::getComponentAbbrev`	`"Sp"`
`NMSNRSGetComponentTitle/ NMSNRS::getComponentTitle`	`"ServiceParametersProv"`
`NMSNRSGetComponentValue/ NMSNRS::getComponentValue`	`"$"`
`NMSNRSGetComponentLevel/ NMSNRS::getComponentLevel`	`3`
`NMSNRSGetCurrentFilePath/ NMSNRS::getCurrentFilePath`	`"/opt/MagellanNMS/data/ nrs/data/ ppc.NODER16.2105.t31,full ,243.800101.data"`

- *void* **NMSNRSResetAttributes**(
  EPIInterface *nrs*); (C)
  void **NMSNRS**::**resetAttributes**(); (C++)

  EPINRSAV * **NMSNRSGetFirstAttribute**(
  EPIInterface nrs); (C)
  EPINRSAV * **NMSNRS**::**getFirstAttribute**(); (C++

  EPINRSAV * **NMSNRSGetNextAttribute**(
  EPIInterface nrs); (C)
  EPINRSAV * **NMSNRS**::**getNextAttribute**(); (C++)

  EPINRSAV * **NMSNRSFindAttribute**(
  EPIInterface nrs,
  char * name); (C)

EPINRSAV * **NMSNRS**::**findAttribute**(
char * name); (C++)

These routines provide access to the attributes of the last fetched component. The first three provide a means to scan the list of attributes, the last one lets you locate a named attribute in the list. The search in case-insensitive. For Passport, both the full attribute name and the prompt can be used. The attribute information is returned as a pointer to a structure containing the following fields:

char * **name**
is the attribute's type; a string for DPN, a numeric attribute ID for Passport (for example, "`LOADPETYPE`" for DPN, `8669` for Passport).

char * **value**
the attribute's value

int **field**
is the attribute's field index (can be used in schema queries described below).

Do not delete or modify the contents of these buffers.
See the description of `NMSNRSFetchNextComponent/ NMSNRS::fetchNextComponent` for an example of use.

Example

The following table shows the possible attribute results from a fetched component (the name in paranthesis is the corresponding attribute title from the component type schema and is indicated here for clarity):

name	value	Field
`_COMPONENT (Component)`	`"ppe/8664"`	`0`
`_HIERARCHY_LEVEL (Hierarchy_level)`	`"3"`	`1`
`_2 (EM)`	`"NODER16"`	`2`
`_279 (FrameRelayUni)`	`"132"`	`3`
`_302 (DataLinkConnectionIdentifier)`	`"206"`	`4`

```
_8664                          “$”          5
(ServiceParametersProv)
OAM                            “OWNER_IWS”  6
(Ownership)
8667                           “2100”       7
(maximumFrameSize)
8668                           “on”         8
(rateEnforcement)
8669                           “64000”      9
(committedInformationRate)
8670                           “64000”      10
(committedBurstSize)
8671                           “0”          11
(excessBurstSize)
8672                           “0”          12
(measurementInterval)
8673                           “off”        13
(rateAdaptation)
8675                           “on”         14
(accounting)
8674                           “7”          15
(raSensitivity)
5997                           “off”        16
(updateBCI)
```

- **EPINRSSchemaComp NMSNRSLoadSchemaComp**(
  EPIInterface nrs,
  char * name,
  int withNext,
  int withSubcomps,
  int withParents); (C)

  EPINRSSchemaComp NMSNRS::**loadSchemaComp**(
  char * name,
  int withNext = 0,
  int withSubcomps = 1,
  int withParents = 1); (C++)

  This routine provides access to the schema (RDF) information. Its arguments are similar to the `NMSNRSIncludeCompType/ NMSNRS::includeCompType` routine. The component types examined by this routine are not added to the list of reported types though (use

`NMSNRSIncludeCompType/NMSNRS::includeCompType` for this). The return value is an opaque pointer suitable for use with the schema access routines described below.

- **EPINRSSchemaComp NMSNRSGetFirstSchemaComp**(
  EPIInterface nrs); (C)
  EPINRSSchemaComp NMSNRS::**getFirstSchemaComp**(); (C++)

  EPINRSSchemaComp NMSNRSGetNextSchemaComp(
  EPIInterface nrs); (C)
  EPINRSSchemaComp NMSNRS::**getNextSchemaComp**(); (C++)

  EPINRSSchemaComp NMSNRSFindSchemaComp(
  EPIInterface nrs,
  char * name); (C)
  EPINRSSchemaComp NMSNRS::**findSchemaComp**(
  char * name); (C++)
  These routines lets you scan the list of component schemas currently known by EPI (not just those to be reported on which can be listed with the `NMSNRSGetFirstReportComp/NMSNRS::getFirstReportComp` and `NMSNRSGetNextReportComp/NMSNRS::getNextReportComp` routines).
  The name provided to the `NMSNRSFindSchemaComp/NMSNRS::findSchemaComp` routine is a string for DPN and a numeric for Passport (the default device type prefix from `NRS.cfg` is used if one is not specified).
  The return value is an opaque pointer suitable for use with the schema access routines described next.

- *char* * **NMSNRSGetSchemaName**(
  EPInterface nrs,
  EPINRSSchemaComp compP); (C)
  char * **NMSNRS::getSchemaName**(
  EPINRSSchemaComp compP = *NULL*); (C)

  char * **NMSNRSGetSchemaTitle**(
  EPInterface nrs,
  EPINRSSchemaComp compP); (C)
  char * **NMSNRS::getSchemaTitle**(

EPINRSSchemaComp compP = *NULL*); (C)

char * **NMSNRSGetSchemaAbbrev**(
EPInterface nrs,
EPINRSSchemaComp compP); (C)
char * **NMSNRS::getSchemaAbbrev**(
EPINRSSchemaComp compP = *NULL*); (C)

int **NMSNRSGetSchemaNbFields**(
EPInterface nrs,
EPINRSSchemaComp compP); (C)
int **NMSNRS::getSchemaNbFields**(
EPINRSSchemaComp compP = *NULL*); (C)
These routines extract the schema information associated with the specified component type, respectively;

Name:
is the component's type specified as `<device>/<comp>` where `<device>` is the device type (`ppc`, `ppe`, or `dpn`), and `<comp>` is the component type (a name for DPN, a numerical ID for Passport, for example, `dpn/UTP` for DPN, `ppc/302` for Passport).

Title:
is the component's name (a name for DPN, the full component name, for Passport, for example, "`UTP`" for DPN, "`FrameRelayUni`" for Passport).

Abbrev:
is the component's abbreviation (a name for DPN, the prompt for Passport, for example, "`UTP`" for DPN, "`FrUni`" for Passport).

NbFields:
is the number of attributes the component has.

The schema component also has a list of possible parent and subcomponents as well as a list of possible attribute descriptions, all accessible through the routines described next.

<compP> if a component type reference returned from one of the routines above and below. If it is NULL, the component type of the last fetched component is examined instead.

Example
The following table shows the possible results for the ServiceParametersProv component type)::

Command	Returned value
NMSNRSGetSchemaName/ NMSNRS::getSchemaName	"ppe/8664"
NMSNRSGetSchemaTitle/ NMSNRS::getSchemaTitle	"ServiceParametersProv"
NMSNRSGetSchemaAbbrev/ NMSNRS::getSchemaAbbrev	"Sp"
NMSNRSGetSchemaNbFields/ NMSNRS::getSchemaNbFields	16

- *void* **NMSNRSResetSchemaSubcomps**(
  EPIInterface nrs,
  EPINRSSchemaComp compP); (C)
  void **NMSNRS**::**resetSchemaSubcomps**(
  EPINRSSchemaComp compP = *NULL*); (C++)

  EPINRSSchemaComp NMSNRSGetSchemaFirstSubcomp(
  EPIInterface nrs,
  EPINRSSchemaComp compP); (C)
  EPINRSSchemaComp NMSNRS::**getSchemaFirstSubcomp**(
  EPINRSSchemaComp compP = *NULL*); (C++)

  EPINRSSchemaComp NMSNRSGetSchemaNextSubcomp(
  EPIInterface nrs,
  EPINRSSchemaComp compP); (C)
  EPINRSSchemaComp NMSNRS::**getSchemaNextSubcomp**(
  EPINRSSchemaComp compP = *NULL*); (C++)
  These routines let you scan the list of possible subcomponent types for the specified <compP> component type reference (from one of the calls above). If that type is specified as NULL, the type of last fetched

component is examined. The returned value is itself another component type reference that can be used to extract its information with the routines described above and below.

- *void* **NMSNRSResetSchemaParents**(
  EPIInterface nrs,
  EPINRSSchemaComp compP); (C)
  void **NMSNRS**::**resetSchemaParents**(
  EPINRSSchemaComp compP = *NULL*); (C++)

  EPINRSSchemaComp NMSNRSGetSchemaFirstParent(
  EPIInterface nrs,
  EPINRSSchemaComp compP); (C)
  EPINRSSchemaComp NMSNRS::**getSchemaFirstParent**(
  EPINRSSchemaComp compP = *NULL*); (C++)

  EPINRSSchemaComp NMSNRSGetSchemaNextParent(
  EPIInterface nrs,
  EPINRSSchemaComp compP); (C)
  EPINRSSchemaComp NMSNRS::**getSchemaNextParent**(
  EPINRSSchemaComp compP = *NULL*); (C++)
  These routines are similar to the previous ones but are used for the list of possible parent component types of the specified type. The type of the last fetched component is examined if `<compP>` is `NULL`.

- *int* **NMSNRSGetSchemaFieldByName**(
  EPIInterface nrs,
  char * name,
  EPINRSSchemaComp compP); (C)
  int **NMSNRS**::**getSchemaFieldByName**(
  char * name,
  EPINRSSchemaComp compP = *NULL*); (C++)

  int **NMSNRSGetSchemaFieldByTitle**(
  EPIInterface nrs,
  char * title,
  EPINRSSchemaComp compP); (C)
  int **NMSNRS**::**getSchemaFieldByTitle**(
  char * title,
  EPINRSSchemaComp compP = *NULL*); (C++)

int **NMSNRSGetSchemaFieldByAbbrev**(
EPIInterface nrs,
char * abbrev,
EPINRSSchemaComp compP); (C)
int **NMSNRS**::**getSchemaFieldByAbbrev**(
char * abbrev,
EPINRSSchemaComp compP = *NULL*); (C++)
These routines let you identify the index for an attribute identified by component type (name), print name (title) or short name (abbreviation). The returned value is a field index suitable to be used in the routines described below.
The type of the last fetched component is examined if `<compP>` is `NULL`.

- *char* * NMSNRSGetSchemaFieldName(
EPIInterface nrs,
int field,
EPINRSSchemaComp compP = *NULL*); (C)
char * NMSNRS::getSchemaFieldName(
int field,
EPINRSSchemaComp compP = *NULL*); (C++)

  char * NMSNRSGetSchemaFieldTitle(
EPIInterface nrs,
int field,
EPINRSSchemaComp compP = *NULL*); (C)
char * NMSNRS::getSchemaFieldTitle
int field,
EPINRSSchemaComp compP = *NULL*); (C++)

  char * NMSNRSGetSchemaFieldAbbrev(
EPIInterface nrs,
int field,
EPINRSSchemaComp compP = *NULL*); (C)
char * NMSNRS::getSchemaFieldAbbrev
int field,
EPINRSSchemaComp compP = *NULL*); (C++)

  char * NMSNRSGetSchemaFieldType(
EPIInterface nrs,

int field,
EPINRSSchemaComp compP = *NULL*); (C)
char * NMSNRS::getSchemaFieldType
int field,
EPINRSSchemaComp compP = *NULL*); (C++)

char * NMSNRSGetSchemaFieldWidth(
EPIInterface nrs,
int field,
EPINRSSchemaComp compP = *NULL*); (C)
char * NMSNRS::getSchemaFieldWidth(
int field,
EPINRSSchemaComp compP = *NULL*); (C++)

char * NMSNRSGetSchemaFieldGroup(
EPIInterface nrs,
int field,
EPINRSSchemaComp compP = *NULL*); (C)
char * NMSNRS::getSchemaFieldGroup
int field,
EPINRSSchemaComp compP = *NULL*); (C++)

These routines provide access to the field information for the component schema specified by `<compP>` (the type of the last fetched component if `NULL`). The returned information is, respectively;

Name:
is the attribute type name for DPN and the attribute numeric ID for Passport.

Title:
is the indexed field name (a string for DPN, the full attribute name for Passport).

Abbreviation:
is the indexed field abbreviation (same as the title for DPN, the attribute prompt for Passport).

Type:
is the indexed field NRS type (`BIT_STRING`, `BOOLEAN`, `DNA`,

HEXADECIMAL, INVISIBLE, LISTINDEX, NOKEY, NUMERIC, or STRING).

Width:
is the indexed field maximum NRS width.

Group:
is the indexed field attribute group type.

Example
The following table shows the possible results for the maximumFrameSize attribute type from the serviceParametersProv component::

Command	Returned value
NMSNRSGetSchemaFieldName/ NMSNRS::getSchemaFieldName	"8667"
NMSNRSGetSchemaFieldTitle/ NMSNRS::getSchemaFieldTitle	"maximumFrameSize"
NMSNRSGetSchemaFieldAbbrev/ NMSNRS::getSchemaFieldAbbrev	"n203"
NMSNRSGetSchemaFieldType/ NMSNRS::getSchemaFieldType	"NUMERIC"
NMSNRSGetSchemaFieldWidth/ NMSNRS::getSchemaFieldWidth	"4"
NMSNRSGetSchemaFieldGroup/ NMSNRS::getSchemaFieldGroup	"ServiceParametersProv"

Sample C and C++ programs

This section provides three sample C/C++ EPI programs: Alarm logging, a synchronous and an asynchronous version, and Passport Card inventory.

Synchronous Alarm logging C EPI example

The following example illustrates the use of the Generic API interface, more precisely the Alarm&Status API, with C EPI. The program simply registers an Alarm&Status API interface to the service selected GMDR server. It then

creates an alarm sieve to receive all alarms. It then logs those alarms with their key fields to a file taking care to close and open a new file when a certain number of alarms have been written to it.

The example source code can be found in the Preside Multiservice Data Manager (MDM) load as `/opt/MagellanNMS/cfg/macros/nms/src/AlarmLogger.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

/* Include the C EPI definitions
 */
#include <EPIPublic.h>

int count = 0;
FILE * outFile = NULL;

/* this function simply (re)opens a log file to
 * spool further alarms
 */
void
openLogFile(froot)
char * froot;
{
    char tname[16];
    char fname[256];
    time_t t = time(NULL);
    struct tm * ts = localtime(&t);

    if (outFile)
        fclose(outFile);

    /* put a time-stamp in the log file name
     */
    strftime(tname, 15, "%y%m%d-%H%M%S", ts);
    sprintf(fname, "%s%s.log", froot, tname);
    outFile = fopen(fname, "w+");
    if (outFile == NULL)
        exit(1);
}
```

```
/* the main routine itself
 */
int
main(int argc, char ** argv)
{
    int maxCount;          /* maximum alarms per file */
    char * froot;          /* file name root */
    EPIInterface api;      /* the NMS API Interface */
    NMSGMDRAPIAlarm * alarm = NULL;
                           /* a received alarm record */

    /* extract the max and file name root from the
     * command line arguments
     */
    maxCount = (argc > 1) ? atoi(argv[1]) : 500;
    froot = (argc > 2) ? argv[2]
          : "/opt/MagellanNMS/data/AlarmLog";

    /* Initialize, Connect, and Register the
     * Alarm&Status API Interface (service selected).
     * (Dont' bother with error checking yet.)
     */
    NMSEPIInit("AlarmLogger");
    api = NMSGMDRAPIInit();
    NMSGMDRInitConnect(api, EPI_GMDR_SELECTED_HOST);
    NMSAPIRegister(api, "AlarmLogger", NULL, NULL);

    /* create an alarm sieve and ask for all attributes
     */
    if ( NMSGMDRAPICreateAlarmSieve(api, 1, NULL, NULL)
                != EPI_SUCCESS )
    {
        /* no sieve (no connection?), then nothing to
         * do.
         */
        exit(1);
    }

    /* open the first log file
     */
    openLogFile(froot);

    /* For each received alarm
     */
    while ( NMSGMDRAPIRecvAlarm(api, &alarm,
```

```
                    EPI_TIMEOUT_FOREVER)
                    == EPI_SUCCESS )
    {
        /* get the NMS display format for the alarm
         */
        char * alarmF;
        alarmF = NMSGMDRAPIFormatAlarm(api,
                        EPI_ALARM_FULL_FORMAT);
        if (alarmF)
        {
            /* if we reached the maximum for a file,
             * open up another one.
             */
            if (++count > maxCount)
                openLogFile(froot);

            /* print the alarm preceeded by its key
             * fields and its length
             */
            fprintf(outFile,
                "%s\n%s\n%s\n%s\n%s\n%d\n%s\n",
                alarm->compId ? alarm->compId : "-",
                alarm->time ? alarm->time : "-",
                alarm->severity ? alarm->severity
                                : "-",
                alarm->event ? alarm->event : "-",
                alarm->faultCode ? alarm->faultCode
                                 : "-",
                strlen(alarmF), alarmF);
        }
    }

}
```

Using the Sun C SparcCompiler, the program was compiled with the following command line:

```
/opt/SUNWspro/bin/cc -znodefs -o AlarmLogger \
        -I/opt/MagellanNMS/lib \
        /opt/MagellanNMS/cfg/macros/\
nms/src/AlarmLogger.c \
        -L/opt/MagellanNMS/lib
```

```
-R /opt/MagellanNMS/lib -lEPIPublic \
-L/usr/openwin/include/X11 \
-R /usr/openwin/include/X11 -lXt -lX11 -lC
```

Asynchronous Alarm logging C++ EPI example

This example is basically the same one as the preceding one except that it is written using the C++ class mappings and uses an asynchronous callback technique to received the alarms.

The example source code can be found in the Preside Multiservice Data Manager (MDM) load as `/opt/MagellanNMS/cfg/macros/nms/src/AlarmLogger.cxx`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

/* Include the C EPI definitions
 */
#include <EPIPublic.hxx>

int count = 0;
FILE * outFile = NULL;
int maxCount;        /* maximum alarms per file */
char * froot;        /* file name root */

/* this function simply (re)opens a log file to
 * spool further alarms
 */
void
openLogFile(froot)
char * froot;
{
    char tname[16];
    char fname[256];
    time_t t = time(NULL);
    struct tm * ts = localtime(&t);

    if (outFile)
        fclose(outFile);
```

```
    /* put a time-stamp in the log file name
     */
    strftime(tname, 15, "%y%m%d-%H%M%S", ts);
    sprintf(fname, "%s%s.log", froot, tname);
    outFile = fopen(fname, "w+");
    if (outFile == NULL)
        exit(1);
}

void
alarmCallback(void * interf, char * ptr,
    int val, int status)
{
    NMSGMDRAPI * api = (NMSGMDRAPI*)interf;

    /* get the NMS display format for the alarm
     */
    char * alarmF = api->formatAlarm(
                EPI_ALARM_FULL_FORMAT);
    EPIGMDRAPI_Alarm * alarmP = api->extractAlarm();
    if (alarmF && alarm)
    {
        /* if we reached the maximum for a file,
         * open up another one.
         */
        if (++count > maxCount)
            openOutFile(froot);

        /* print the alarm preceeded by its key
         * fields and its length
         */
        fprintf(outFile,
            "%s\n%s\n%s\n%s\n%s\n%d\n%s\n",
            alarmP->compId ? alarmP->compId : "-",
            alarmP->time ? alarmP->time : "-",
            alarmP->severity ? alarmP->severity
                        : "-",
            alarmP->event ? alarmP->event : "-",
            alarmP->faultCode ? alarmP->faultCode
                          : "-",
            strlen(alarmF), alarmF);
    }
}
```

```
/* the main routine itself
 */
int
main(int argc, char ** argv)
{
    /* extract the max and file name root from the
     * command line arguments
     */
    maxCount = (argc > 1) ? atoi(argv[1]) : 500;
    froot = (argc > 2) ? argv[2]
          : "/opt/MagellanNMS/data/AlarmLog";

    /* Initialize, Connect, and Register the
     * Alarm&Status API Interface (service selected).
     * (Dont' bother with error checking yet.)
     */
    NMSEPIInit("AlarmLogger");
    NMSGMDRAPI * api = new NMSGMDRAPI();
    api->connect(EPIGenAPI_GMDR_SERVICE, "localhost");
    api->registerAPI("AlarmLogger", NULL, NULL);

    /* create an alarm sieve and ask for all attributes
     */
    if ( api->createAlarmSieve(1, NULL, NULL)
                != EPI_SUCCESS )
    {
        /* no sieve (no connection?), then nothing to
         * do.
         */
        exit(1);
    }

    /* open the first log file
     */
    openLogFile(froot);

    /* bind the alarm receiving callback
     */
    api->bindCallback(alarmCallback);

    /* jump into the event loop
     */
    NMSEPIEventLoop();

    return 0;
```

```
}
```

Using the Sun C++ SparcCompiler, the program was compiled with the following command line:

```
/opt/SUNWspro/bin/CC -qoption ld -znodefs \
        -o AlarmLogger \
        -I/opt/MagellanNMS/lib \
        /opt/MagellanNMS/cfg/macros/\
nms/src/AlarmLogger.cxx \
        -L/opt/MagellanNMS/lib
        -R /opt/MagellanNMS/lib -lEPIPublic \
        -L/usr/openwin/include/X11 \
        -R /usr/openwin/include/X11 -lXt -lX11
```

Passport Card inventory C++ EPI example

The following C++ program uses the EPI C++ command interface to produce a card inventory of a specified Passport node (using the current user session). This example is available in the Preside Multiservice Data Manager (MDM) load in `/opt/MagellanNMS/cfg/macros/nms/src/PPCardInv.cxx`.

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

// Include the C++ EPI definitions
//
#include <EPIPublic.hxx>

// Main routine
//
int
main(int argc, char ** argv)
{
    // Extract the arguments.
    //
    if ( argc < 3 )
    {
```

```
        cerr << "ppcardrep <group> <passport>" << endl;
        exit(1);
    }

    char * grp = argv[1];
    char * mod = argv[2];

    // Initializes the command interface.
    //
    NMSEPIInit("PPCard");
    NMSCmd * cmd = new NMSCmd();

    // Connect to the current session servers.
    //
    if ( cmd->connect() != EPI_SUCCESS )
    {
        cerr << "\n*** Failed to connect to the \
command Session servers." << endl;
        exit(1);
    }

    cout << "\n Passport Card Inventory\n"
         <<
"----------------------------------------\
----------------------------\n"
        << "Node         Card Type         \
Inserted     Serial #       Firm. Rev.     LP\n"
        << "------------ ---- ------------ \
------------ -------------- -------------- ------"
        << endl;


    // Do we have access to the node.
    //
    char cmdline[128];
    sprintf(cmdline, "%s h -v(d) shelf card", mod);
    cmd->sendCommand(grp, cmdline);
    char * dontcare;
    cmd->recvFullReply(&dontcare);
    if ( cmd->patternMatch(&dontcare, "Shelf Card")
                != EPI_SUCCESS )
    {
        cerr << "\n*** Passport node " << mod
             << " does not seem to be reachable."
```

```
            << endl;
        exit(1);
    }

    // Does the node support tabular output
    //
    char * notab = "";
    if ( cmd->patternMatch(&dontcare, "noTabular" )
                == EPI_SUCCESS )
        notab = "-noTabular";

    // Get the number of slots from the shelf component.
    //
    int numberOfSlots = 16;
    sprintf(cmdline, "%s d %s shelf numberOfSlots",
                mod, notab);
    cmd->sendCommand(grp, cmdline);
    char * line;
    while ( cmd->recvNextLine(&line) == EPI_SUCCESS )
    {
        cmd->getColumn(&line, 1);
        if ( strcasecmp(line, "numberOfSlots") == 0)
        {
            cmd->getColumn(&line, 3);
            numberOfSlots = atoi(line);
        }
    }

    // Get the needed attributes from all card
    // components.
    //
    char card[64];
    char cardType[64];
    char insertedCardType[64];
    char serialNumber[64];
    char activeFirmwareVersion[64];
    char currentLp[64];
    cardType[0] = '\0';
    insertedCardType[0] = '\0';

    sprintf(cmdline, "%s d %s shelf card/* \
cardType,insertedCardType,serialNumber,\
activeFirmwareVersion,currentLP",
            mod, notab);
```

```
cmd->sendCommand(grp, cmdline);
while ( cmd->recvNextLine(&line) == EPI_SUCCESS )
{
    char col[64], val[64];
    cmd->getColumn(&line, 1);
    strcpy(col, line ? line : "");
    cmd->getColumn(&line, 3);
    strcpy(val, line ? line : "");

    if ( strcasecmp(col, "cardType") == 0 )
        strcpy(cardType, val);
    else if ( strcasecmp(col, "insertedCardType")
              == 0 )
        strcpy(insertedCardType, val);
    else if ( strcasecmp(col,
                  "activeFirmwareVersion") == 0 )
        strcpy(activeFirmwareVersion, val);
    else if ( strcasecmp(col, "serialNumber")
              == 0 )
        strcpy(serialNumber, val);
    else if ( strcasecmp(col, "currentLp") == 0 )
        strcpy(currentLp, val);
    else if ( strcasecmp(col, "Shelf") == 0 )
    {
        // This is the name of a card (either the
        // first or another one in the list).
        //
        if (   ( cardType[0] != '\0' )
            && ( insertedCardType[0] != '\0' )
            && (   (strcasecmp(cardType, "none")
                     != 0)
                || (strcasecmp(insertedCardType,
                         "none") != 0) ) )
        {
            // Print the info on the preceding
            // card.
            //
            cout << setiosflags(ois::left)
                 << setw(12) << mod
                 << setw(4)  << card
                 << setw(12) << cardType
                 << setw(12) << insertedCardType
                 << setw(14) << serialNumber
```

```
                        << setw(14)
                           << activeFirmwareVersion
                        << setw(0)  << currentLp
                        << endl;
                    cardType[0] = '\0';
                    insertedCardType[0] = '\0';

                }

                // Extract the card number from the name
                // and stop if maximum.
                //
                cmd->getColumn(&line, 2);
                line = NMSEPIPatternMatch("Card/", line,
                                          "", 0);
                strcpy(card, line);
                if ( atoi(card) > numberOfSlots )
                    break;
            } // if ... else if ...
        } // while

        // Print the last card's information if required.
        //
        if (   ( cardType[0] != '\0' )
            && ( insertedCardType[0] != '\0' )
            && (   (strcasecmp(cardType, "none") != 0)
                || (strcasecmp(insertedCardType, "none")
                    != 0) ) )
        {
            cout << setiosflags(ios::left)
                 << setw(12) << mod
                 << setw(4)  << card
                 << setw(12) << cardType
                 << setw(12) << insertedCardType
                 << setw(14) << serialNumber
                 << setw(14) << activeFirmwareVersion
                 << setw(0)  << currentLp
                 << endl;
        }

        exit(0);
    }
```

Using the Sun C++ SparcCompiler, the program was compiled with the following command line:

```
/opt/SUNWspro/bin/CC -qoption ld -znodefs \
        -o PPCardInv \
        -I/opt/MagellanNMS/lib \
        /opt/MagellanNMS/cfg/macros/\
nms/src/PPCardInv.cxx \
        -L/opt/MagellanNMS/lib
        -R /opt/MagellanNMS/lib -lEPIPublic \
        -L/usr/openwin/include/X11 \
        -R /usr/openwin/include/X11 -lXt -lX11
```

Chapter 6
CORBA Embedded Programming Interface

This section describes the CORBA version of the Embedded Programming Interface (EPI). This chapter contains the following:

Code conventions

There are two code conventions used in this chapter:

- \
  A back slash (\) indicates that the line of code is continued on the next line space.
- //
  A message line that starts with a double-slash is treated as a comment (C++ convention). For C code, comments are also indicated with a pair of /* */ delimiters.
- Except where indicated, all examples are specified using C++.

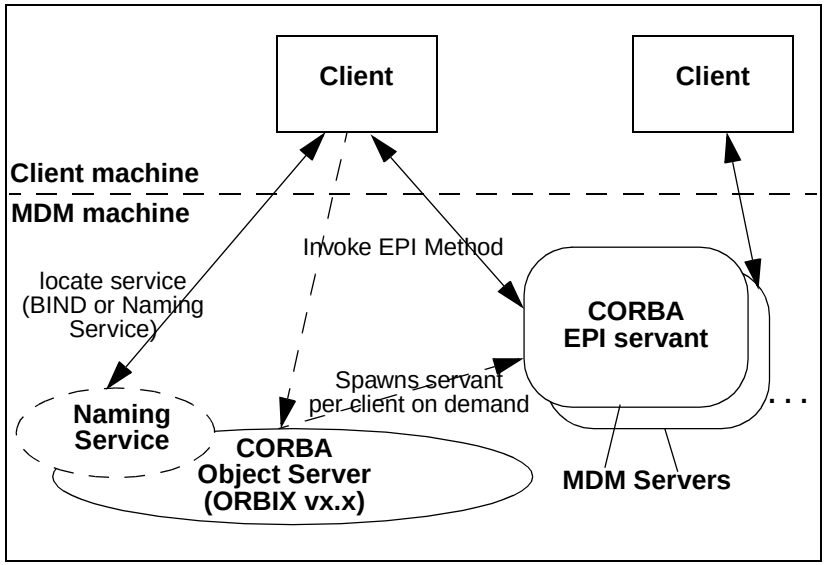
Integration methodology

The CORBA version of the EPI interface is a language-independent remotable interface. It allows you to use the EPI capabilities at a programatic level without having to develop and run an EPI client on the MDM workstation. (EPI requires local access to the MDM shared libraries and communication infrastructure.) This allows a broader use of the EPI capabilities but it also has a few impacts on the way these capabilities are expressed. Most notable is the fact that CORBA EPI operations will return more information directly rather than provide information for individual accessor functions. This reduces inter-process communication traffic. To do this, the CORBA EPI interface uses return-by-value SEQUENCE type attributes.

Like other CORBA interfaces, CORBA EPI interface consists of a specialized process on the MDM machine (the servant) and a set of portable and language-independent interface descriptions (the CORBA EPI IDLs). The interface supported by the CORBA EPI servant does not align with a particular current or discussed CORBA-based standard. It does provide MDM access capabilities that are similar to those provided by other EPI interfaces.

Unlike other EPI interfaces, the CORBA EPI is not enabled by default. Before it can be used, the CORBA EPI servant must be registered in the CORBA Implementation Repository. The CORBA EPI servant is registered by the Orbix Daemon in an instance-per-client-process (PID) mode. This means that a new instance of the servant will be started for each CORBA client process that makes requests to the CORBA EPI servant object. That servant is only accessible to its requesting client process. See “Enabling the CORBA EPI servants” (page 500) for the procedures to enable the CORBA EPI servants. See “Client-server relationship” (page 497).

Figure 3
Client-server relationship



The CORBA EPI servants are located using either the Orbix BIND mechanism or the CORBA standard Naming Service. Using the BIND mechanism, the servants are located on the Orbix/MDM workstation using the appropriate service name. Using the CORBA Naming Service, the servants are located using a fully qualified naming context. See the table “Supported BIND and Naming Service names” (page 498).

Table 1
Supported BIND and Naming Service names

BIND names	Naming Service Names	Servants and interfaces
ntCmdAccessSvr	/ntEPI/ntCmdAccess/ ntCmdAccessSvr	Command Access servant BaseEPI, CmdServant, CmdSession, and CmdInterface interfaces.
ntApiAccessSvr	/ntEPI/ntApiAccess/ ntApiAccessSvr	Generic and specialized API Access BaseEPI, APIServant, GenAPI, GMDRAPI, NMAPI, and HGDSAPI interfaces.

It is possible to add a suffix to the naming service name using the `-m` option for the `setUpCPI.sh` script. When you add a suffix, it will appear at the end of the default name with an underscore as separator. For example, if you added the suffix “west” to the setup script for the API servant shown above, the new service name would be `ntApiAccessSvr_west` for BIND access and `/ntEPI/ntCmdAccess/ntApiAccessSvr_west` for Naming Service access. This technique allows you to register servants on different workstations with cooperating Name Services and still be able to control which workstation will activate the servant.

When a reference is requested for the named servant, Orbix automatically spawns a new instance of its process for private use by the calling client. From that point on, the various interface methods can be invoked in the usual manner for a CORBA-based interface. The servant will automatically terminate once it detects that its client connection is lost. The servant supports no persistency or automatic passivation.

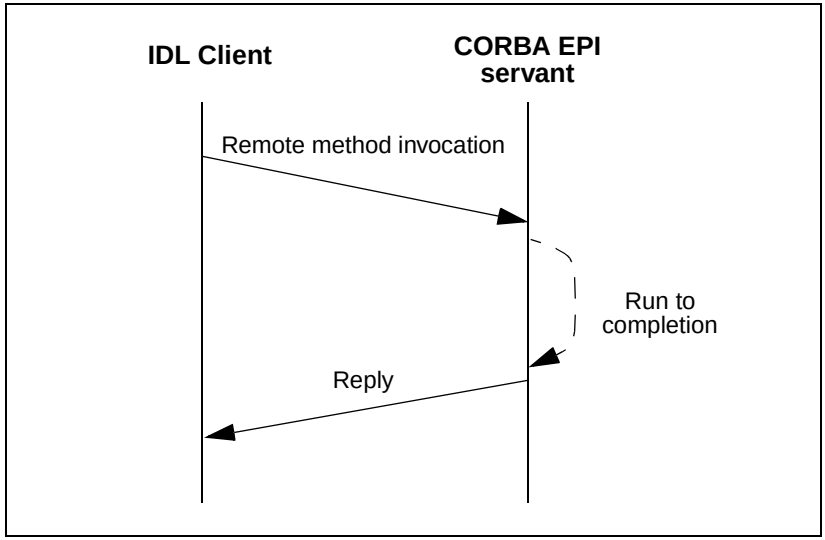
See “Locating an EPI servant reference” on page 506 for more details on obtaining references to CORBA EPI servants.

Note: The examples that follow assume that a servant reference has already been obtained.

The CORBA EPI interface supports both synchronous and asynchronous modes of operation; however, both modes function differently for a remote interface.

In synchronous mode, remote method calls run to completion and return their results immediately. While the method call is being fulfilled, the servant is blocked on that call so it is impossible to invoke two synchronous methods at once. (The CORBA client may be multi-threaded but the servant still behaves in run-to-completion manner.) Therefore you must be careful when issuing long waiting method calls such as reply reception method calls. This impact is minimized by the fact that there is a specific servant per client instance. See the figure “Synchronous mode remote interface” (page 499)

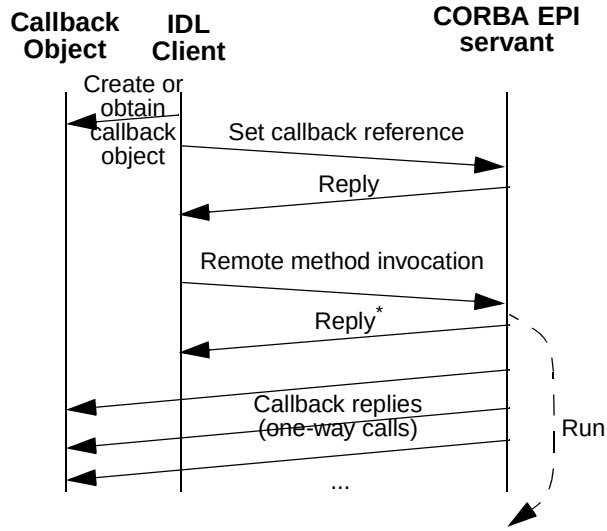
Figure 4
Synchronous mode remote interface



In asynchronous mode, the servant is given an object reference (implemented within the same client or another process) that supports a CORBA EPI specific interface (a callback operation signature). Like non-CORBA EPI interfaces, the callback method on the client is invoked by the servant when

information is available. Note that the callback is a one-way call from the servant's perspective. The client process must be ready to process callbacks as soon as they are available. (This may imply multi-threading the client.) The callback will provide all the received information to the client since it is impossible to query the remote servant for additional information. This is because the remote servant receives, processes, and forwards new replies and notifications while the callback is being processed by the client. (The callback operation is a one-way call.) See the figure "Asynchronous mode remote interface" (page 500).

Figure 5
Asynchronous mode remote interface



* This reply is actually part of the CORBA transport. The actual replied data is sent back to the callback object.

Enabling the CORBA EPI servants

To use the CORBA EPI interface, necessary servant implementation must be registered into the Orbix daemon. The following procedures indicate how this is done using an MDM supplied utility script. Note that the procedure below only needs to be performed once. See the Orbix administration documentation for more detailed background.

See the table "Servant implementation parameters" (page 501) for a summary of the various CORBA EPI servant implementation parameters

Table 2
Servant implementation parameters

Servant names	Servant implementation paths	Servant and interfaces
ntCmdAccessSvr	/opt/MagellanNMS/bin/cmdaccesssrv	Command Access servant BaseEPI, CmdServant, CmdSession, and CmdInterface interfaces.
ntApiAccessSvr	/opt/MagellanNMS/bin/apiaccesssrv	Generic and specialized API Access BaseEPI, APIServant, GenAPI, GMDRAPI, NMAPI, and HGDSAPI interfaces.

Registering the CORBA EPI servants

To register a CORBA EPI servant, execute the following command as root user when the Orbix daemon is running:

```
/opt/MagellanNMS/bin/setupCPI.sh <Orbix path> \
              [-m <unique ID>]
              [no_ns] [enabled_mode] [all] \
              <servants...>
```

Where:

`<Orbix path>` is the root of the Orbix installation directory on this machine.

`-m <unique ID>` gives a specific service name to this servant. The specified name is appended to the default servant name with an underscore character (for example, `-m west`

used to setup the API servant results in a service name of `ntApiAccessSvr_west` for BIND and `/ntEPI/ntApiAccess/ntApiAccessSvr_west` for the COS Naming Service).

`no_ns` indicates that only Orbix BIND naming is used and no Name Service registration is to be attempted. Otherwise (by default) Naming Service access is enabled for the servants.

`enabled_mode` enables remote invocations of UNIX and SNMP commands and setting of workstation-wide context variables. By default, the Command Access servant does not allow remote invocations of UNIX and SNMP commands for security reasons. As well, both the Command and API Access servants do not allow the setting of workstation-wide context variables for similar reasons.

`all` lets any remote client invoke a servant (`chmodit i+all` and `l+all` options of Orbix). By default, the servant is registered in a way that only root users (local and remote clients) can invoke it. With this option, any remote client can invoke a servant.

`<servants...>` are the names of the CORBA EPI servants to be enabled. Currently, the only servant names allowed are `ntCmdAccessSvr` for the CORBA EPI Command Access servant, and `ntApiAccessSvr` for the CORBA EPI API Access servant.

The script performs the necessary steps to register and enable the requested CORBA EPI servants.

> ***Note:*** You only need to execute the commands once unless the supporting workstation information (for example, the IP address) changes.

Refer to your Orbix documentation for more information on servant, implementation and interface administration.

Logging/tracing a servant

For development purposes, the CORBA EPI servants can trace their operations. To enable this, you must create (touch) a servant specific file. For example, to enable the Command Access servant, you must create a file called:

```
/opt/MagellanNMS/data/log/csvr/cmdaccesssrv.trace
```

Any Command servant spawned by the Orbix ORB while this file exists will trace its operations to files (one per servant) named:

```
cmdaccesssrv_<yyy><mm><dd>T<hh><mm><ss>.<pid>.log
```

in the same directory with the month (`mm`), day (`dd`), hour (`hh`), and PID (`pid`) of its creation. Similarly, the control file for the API Access servant is `apiaccesssrv.trace` and the matching log file prefix is `apiaccesssrv_`.

Interface

The CORBA EPI interface provides, at this point, three groups of function calls:

- **Base**:
  Provides mapping to the EPI base routines and utilities.
- **API Access:**
  Provides access to the Preside Multiservice Data Manager (MDM) API open interface capabilities including specialized access to the Alarm&Status API, the Network Model API and the Host Group Directory Service API.
- **Command Access**:
  Provides access to the Preside Multiservice Data Manager (MDM) command macro capabilities.

Command usage information

The following information applies to the provided functions:

- **command argument**
  You need to specify all function arguments in the indicated order.
- **return codes**
  The CORBA EPI IDL provides common function return codes to indicate the successful or failed execution of the call. Functions usually return an enumerated success/failure indication. The possible values for these return codes, as defined in the `EPIBase.idl` IDL (module `nt_EPI_BASE`) as the `EPI_Result` enumeration, follow:

`CEPI_SUCCESS`	success
`CEPI_FAILED`	operation failed
`CEPI_TIMEOUT`	operation timed out
`CEPI_NO_CONNECTION`	interface not created, initialized or connected to server
`CEPI_BUSY`	a query is already active
`CEPI_NO_QUERY`	no active query
`CEPI_END`	end of query (no more replies)
`CEPI_BADARG`	bad parameter passed to function call

In addition, some methods will raise an exception (defined in the `EPIBase.idl` IDL, module `nt_CPI_EXCEPTION`, exception `CPIException`) under specific failure scenarios (see the description of the methods for the details). The exception object contains two data members:

enum **ErrorCodes exceptType**	one of **EPIEX_NO_CONNECTION**, **EPIEX_BUSY**, **CPI_CLT_CALLBACK_OBJ_NOT_VALID**, or **CPI_SERVER_ERROR**. The first two map to the error codes above. The last two represent errors specific to the CORBA interface.
string **exceptMessage**	Textual representation of the error message

- **time-outs**
  Many functions allow you to specify a time-out value indicating the amount of time to wait for a reply before giving up. The following constant values can be used (defined in module `nt_EPI_BASE`):

CEPI_TIMEOUT_FOREVER	the function is blocking and forever waits for a reply.
CEPI_TIMEOUT_POLL	the function does not wait and returns immediately if a reply is not already available.
`other numerical values`	the function waits at most for the specified number of seconds for a reply to become available.

- **returned strings and text**
  Some functions return text string values either directly or as part of structures. These values must be handled, released, and possibly freed when they are no longer needed.
  `NULL` string pointers are not returned to indicate errors by convention for CORBA interfaces. Functions that did so in the C/C++ EPI interface return their strings as `out` parameters instead. In case of errors, an error code value (`EPIResult` of `CEPI_FAILED`) and an empty string are returned.

- **returned SEQUENCEs**
  Some functions return sequences (lists) of structures with replied data from the EPI servant. These data structures are to be manipulated in the usual manner the used CORBA implementation language.

  Note: The example code that is included is written in C++, but other languages could have been used also.

Differences with the C/C++ language EPIs

The CORBA level interface is similar to the C++ language interfaces. The differences are:

- use of exceptions to indicate some error situations.
  See “Command usage information” (page 504).
- replied data mainly returned as `out` value parameters to reduce the number of inter-process method invocations.
- some utility functions and capabilities of the local EPI interfaces are not mapped to the IDLs as their usefulness is limited in a remote/inter-process environment. Most data manipulation functions (such as attribute list scanning and column extraction) have therefore been removed in favor of passing back the entire data at once in the receive and callback calls.
- The Command Access interface’s Flow/Templating calls do not support an asynchronous mode at this point.
- multi-threaded operations on the client side is supported. Since there is no direct MDM code interaction, the MDM limitations do not apply.

Locating an EPI servant reference

The first thing that you must do to use a CORBA EPI interface is to locate a reference to the appropriate EPI servant object. This can be done in one of two ways; using Orbix’ BIND mechanism or using the CORBA Naming Service. See “Enabling the CORBA EPI servants” on page 500 for the instructions on how to set up the servant to support either mode.

Locating an EPI servant using Orbix BIND:

You need to use the static _bind method of the generated stub for the servant code:

```
<servant stub var or ptr> =
    <servant stub class)->_bind(
                <service name>, <supporting host>);
```

For example, locating a Command Access servant with BIND looks like:

```
nt_CMD_ACCESS::CmdServant_var cmdSvrRef;
try {
    cmdvrRef = nt_CMD_ACCESS::CmdServant::_bind(
                "0:ntCmdAccessSrv", "wcars036”);
    if (CORBA::is_nil(cmdSvrRef)) {
                ... error
```

Locating an EPI servant using the CORBA Naming Service:

The first thing that you need to do is to locate the Naming Service object reference through the ORB and narrow it down to the naming root context:

```
try {
    // use this next command only if not yet initialized
    CORBA::ORB_var orb =
                CORBA::ORB_init(argc, argv, "Orbix");
    CORBA::Object_var namingSvrRef =
                orb->resolve_initial_references(
                            "NameService");
    CosNaming::NamingContext_var rootContext =
                CosNaming::NamingContext::_narrow(
                            namingSvrRef);
```

The root context can then be used to locate the EPI servant object:

```
    // ... build the CosNaming Name object to
    // ... identify the service to lookup
    CosNaming::Name_var object_name
                = new CosNaming::Name(<nb levels>);
    // .. assign the various level names
    CORBA::Object_var obj =
                rootContext->resolve(object_name);
    <servant stub var> <servant reference> =
```

```
            <servant stub class>::_narrow(obj);
if (CORBA::is_nil(<servant reference>)) {
            ... error
```

For example, to locate the a Command Access servant with the CORBA Naming Service, use:

```
CosNaming::Name_var object_name
            = new CosNaming::Name(3);
object_name->length(3);
object_name[0].id = CORBA::string_dup("ntEPI");
object_name[0].kind
            = CORBA::string_dup("");
object_name[1].id
            = CORBA::string_dup("ntCmdAccess");
object_name[1].kind
            = CORBA::string_dup("");
object_name[2].id
            = CORBA::string_dup("ntCmdAccessSvr");
object_name[2].kind
            = CORBA::string_dup("");
CORBA::Object_var obj =
            rootContext->resolve(object_name);
nt_CMD_ACCESS::CmdServant_var cmdSvrRef =
            nt_CMD_ACCESS::CmdServant::_narrow(
                        obj);
if (CORBA::is_nil(cmdSvrRef)) {
            ... error
```

Since the servant is registered as a per-client process, both these techniques this will cause a new servant process to be started on the servant host and a reference to its matching object to be returned.

Base

The Base CORBA EPI interface is defined in the IDL file:

```
/opt/MagellanNMS/lib/idl/EPIBase.idl
```

This IDL file is also required by all the other CORBA EPI IDLS.

In addition, the

```
/opt/MagellanNMS/lib/idl/EPIContextCallback.idl
```

IDL file describes the client-side interface to be implemented in order to support asynchronous Context Server interaction if it is to be used.

This IDL defines two modules: `nt_CPI_EXCEPTION` and `nt_EPI_BASE`. The module `nt_CPI_EXCEPTION` contains the returned exception information and the module `nt_EPI_BASE` contains base utilities and the definition of the `BaseEPI` interface implemented by all CORBA EPI servants.

Common Operations

Interface: `BaseEPI`
Module: `nt_EPI_BASE`
IDL file: `EPIBase.idl`

The following operations are defined as part of the `BaseEPI` interface:

- **nt_EPI_BASE**::**EPIResult** res =
  nt_EPI_BASE::**BaseEPI**::**convertCompId**(
  in *string* compId,
  in **nt_EPI_BASE**::**BaseEPI**::**CompIdConversionType** conversion,
  out *string* outCompId);

  Converts the specified component ID, and returns the string result in `<outCompId>` and a value of `nt_EPI_BASE::EPI_SUCCESS` on success, an empty string (`""`) and a return value of `nt_EPI_BASE::CEPI_FAILED` on failure. The conversion codes, `<conversion>`, are defined in the same module as the values of the `CompIdConversionType` enumeration as follows:

`CEPI_CANON_CVT`	converts to canonical API format (for example, "`PM AM1 PE 1 PI 1`").
`CEPI_DISPLAY_CVT`	converts to display format (for example, "`PM/AM1 PE/1 PI/1`").
`CEPI_TYPE_CVT`	extracts the module/link type (for example, "`PM`" or "`NL`").
`CEPI_MNEMONIC_CVT`	extracts the module mnemonic (for example, "`AM1`").

CEPI_EP1_CVT and **CEPI_EP2_CVT**	extract the first and second link endpoints in canonical API format.
CEPI_DPN_CVT	converts a DPN-100 OA or a PE/PI/PO component ID to a form suitable for commands (`<mnemonic> [pe <pe#>|<pi #> [<po #>]]`). (For example, "`AM1 11`" is the output for an input of "`PM/AM1 PE/11 PI/11`".)
CEPI_SWITCH_CVT	returns the module-level component ID in canonical API format (for example, "`PM AM1`" for an input of "`PM/AM1 PE/11`").
CEPI_OMNI_CVT	returns the Preside Multiservice Data Manager (MDM) HP-OpenView DeskTop compatible component ID (similar to display format except for link names).

Examples

```
nt_CMD_ACCESS::CmdServant_var epiSvrRef;
// ... locate servant reference
CORBA::String_var ep1;
if ( epiSvrRef->convertCompId(linkId,
                              nt_EPI_BASE::CEPI_EP1_CVT,
                              ep1)
                == nt_EPI_BASE::CEPI_SUCCESS ) {
   printf("Endpoint1: %s\n", (const char*)ep1);

CORBA::String_var dpn;
if ( epiSvrRef->convertCompId(portId,
                              nt_EPI_BASE::CEPI_DPN_CVT,
                              dpn)
                == nt_EPI_BASE::CEPI_SUCCESS ) {
   sprintf(cmd, "%s enable", (const char*)dpn);
   cmdIfRef->sendCommand(cmd);
```

- *short* compare = **nt_EPI_BASE**::**BaseEPI**::**compareCompIds**(
  in string comp1,
  in string comp2);

Compares the two component IDs and returns `0` if they are identical, `<0` if the first component ID precedes the second, and `>0` if the second precedes the first. This is an intelligent comparison that is aware of differences in order between numerical and textual instance value sorting.

- **nt_EPI_BASE**::**EPIResult** res =
  nt_EPI_BASE::**BaseEPI**::**convertTime**(
  in *string* inTime,
  in **nt_EPI_BASE**::**BaseEPI**::**TimeConversionType** cvt,
  in *boolean* isSTC,
  in *string* param,
  out string outTime);

  Converts the specified time string, `inTime`, and returns the resultant string in `<outTime>` (with a result of `nt_EPI_BASE::CEPI_SUCCESS`) on success or an empty string (`“”` with a result of `nt_EPI_BASE::EPI_FAILED`) on failure. The input time string can be in the following formats:

  API (`YYYY MM DD HH MM SS`)
  Common Alarm (`YY-MM-DD HH:MM:SS`)
  UNIX Epoch (`<number of seconds since 1970>`)
  Passport reply (`YYYY-MM-DD HH:MM_SS`)

  The returned time is in the same time frame as the input one except when `<isSTC>` is set to `True` or when the `CEPI_STC_TCVT` conversion is performed. If `<isSTC>` is set to `True`, the input time is assumed to be in Standard Time Coordinates, that is, Greenwich Mean Time (GMT) If `<isSTC>` is set to `True` value and `CEPI_STC_TCVT` is not used, then the output time is converted from STC to local workstation time. If the input time is an empty string, the current workstation time is used and `<isSTC>` is ignored.

  The conversions are as follows:

CEPI_API_TCVT	produces the time in API format
CEPI_STC_TCVT	produces the time in API format converted to Standard Time Coordinates
CEPI_EPOCH_TCVT	returns the time as a UNIX Epoch value (number of seconds since 1970)
CEPI_FTIME_TCVT	returns the time in the format specified as the `<param>` argument (see `man -s3c strftime`, 100 characters maximum)
CEPI_UNIX_TCVT	returns the time as the default time format for the workstation's LOCALE (see `man -s3c ctime`)
CEPI_ALARM_TCVT	returns the time in Common Alarm format
CEPI_PASSPORT_TCVT	returns the time in Passport reply format
CEPI_DAY_OFFSET_TCVT	returns the time in API format after applying the positive or negative offset in days specified in the `<param>` string
CEPI_SECONDS_OFFSET_TCVT	returns the time in API format after applying the specified positive or negative offset in seconds specified in the `<param>` string

CEPI_MIDNIGHT_OFFSET_TCVT returns the number of seconds from the previous midnight and the specified time (or current time)

CEPI_STC_OFFSET_TCVT returns the number of seconds between the servant workstation time and the coordinated universal time (UTC) and, if needed, taking daylight savings time into consideration. The offset is positive going west from UTC (same as UNIX timezone/altzone)

Examples

```
CORBA::String_var tim;
if ( epiSvrRef->convertTime("",
              nt_EPI_BASE::CEPI_DAY_OFFSET_TCVT,
                        False, "7", tim)
              == nt_EPI_BASE::CEPI_SUCCESS ) {
   printf("%s\n", (const char*)tim);
```

produces something like (the time 7 days ago):

```
2000 03 24 17 01 19
```

```
if ( epiSvrRef->convertTime("2000 03 24 17 01 19",
              nt_EPI_BASE::CEPI_EPOCH_TCVT,
                        False, "", tim)
              == nt_EPI_BASE::CEPI_SUCCESS ) {
   printf("%s\n", (const char*)tim);
```

produces something like (the same as a UNIX epoch):

```
953935279
```

```
if ( epiSvrRef->convertTime("953935279",
              nt_EPI_BASE::CEPI_FTIME_TCVT,False,
              "Time was: %a %b %d %l:%M%p", tim)
              == nt_EPI_BASE::CEPI_SUCCESS ) {
   printf("%s\n", (const char*)tim);
```

produces something like (the same using a custom format):

```
Time was: Fri Mar 24  5:01PM
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_EPI_BASE::**BaseEPI**::**patternMatch**(
  in *string* pattern,
  in *string* target,
  out string outMatched);

  nt_EPI_BASE::**EPIResult** res =
  nt_EPI_BASE::**BaseEPI**::**patternSubstitute**(
  in *string* pattern,
  in *string* target,
  in *string* substitute,
  in *boolean* all,
  out string outSubstituted);

  `pattermMatch` performs pattern matching and substitution of `<pattern>` against `<target>` and returns the string representing the matching part in `<outMatched>` on success or an empty string (“”) on failure. The `<pattern>` is specified in grep syntax. You can include one `\\(` delimited sub-pattern (note the double backslashes needed to work around C/C++’s string escaping mechanism). The text matching the sub-pattern is returned if one is specified instead of the entire match portion. In the `patternSubstitute` form, the matching substring is replaced by the value of `<substitute>` and the resulting string returned as `<outSubstituted>` on success, and empty string (“”) on failure. If `<all>` is `True`, the function substitutes all matching substrings.

  Example

```
// ... extract the PE number
CORBA::String_var res;
if ( epiSvrRef->patternMatch(
              “pe \\(\\[0-9\\]*\\)down”, target,
              res) == nt_EPI_BASE::CEPI_SUCCESS ) {
   printf(“PE# %s is down.\n”, (const char*)res);
...
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_EPI_BASE::**BaseEPI**::**getContext**(
  in **nt_EPI_BASE**::**BaseEPI**::**ContextDomainType**
  domain,
  in *string* contextName,
  out string varValue);

nt_EPI_BASE::**EPI_Result** res =
nt_EPI_BASE::**BaseEPI**::**setContext**(
in **nt_EPI_BASE**::**BaseEPI**::**ContextDomainType** domain,
in string contextName,
in string contextValue)
raises(**nt_CPI_EXCEPTION**::**CPIException**);

Respectively gets or sets an MDM context variable, `<contextName>`, from or into the specified MDM Context Domain, `domain`. `getContext` returns the variable value as `<varValue>` and a return code of `nt_EPI_BASE::CEPI_SUCCESS` on success or an empty string (“”) and a return code of `nt_EPI_BASE::CEPI_FAILED` on failure. `setContext` returns `CEPI_SUCCESS` or `nt_EPI_BASE::CEPI_FAILED`. In addition, `setContext` can raise one of the `CPIExceptions`. The supported Context Domains are (defined in the `EPIBase.idl` IDL, `nt_EPI_BASE` module, `BaseEPI` interface, `ContextDomainType` enum):

`CEPI_USER_CONTEXT`	the current User Session Context
`CEPI_WS_CONTEXT`	the Workstation Wide Context
`CEPI_SERVICE_CONTEXT`	the Service Selection Context

Example

```
CORBA::String_var ctx;
if ( epiSvrRef->getContext(
    nt_EPI_BASE::BaseEPI::CEPI_USER_CONTEXT, \
                            “DPN_QUICK_STEP”, ctx)
              == nt_EPI_BASE::CEPI_SUCCESS ) {
    printf(“Hot context: %s\n”, (const char*)ctx);
```

Note: By default, it is impossible to set context variable in the workstation-wide domain. To allow it, you must configure the CORBA EPI Command servant appropriately as described in “Command servant” on page 547.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_EPI_BASE::**BaseEPI**::**registerContextInterest**(
  in **ClientContextInterestCB** clientContextCBObj,

in **nt_EPI_BASE**::**BaseEPI**::**ContextDomainType** domain,
in **nt_EPI_BASE**::**StringSeq** varnames);

nt_EPI_BASE::**EPI_Result** res =
nt_EPI_BASE::**BaseEPI**::**unregisterContextInterest**();

Binds the specified callback client-side object reference `<clientContextCBObj>` to the Preside Multiservice Data Manager (MDM) Context variables identified by `<contextName>` (`StringSeq` is defined in the module as a typedef to `sequence<string>`) and `<domain>` (see `getContext` above). This object implements the `ClientContextInterestCB` interface (defined in the `EPIContextCallback.idl` IDL) which defines a single one-way callback function invoked by the servant whenever one of the named variable changes value. The callback function has the following signature:

Interface: `ClientContextInterestCB`
IDL file: `EPIContextInterfaceCB.idl`

oneway void
ClientContextInterestCB::**contextInterestCB**(
in **nt_EPI_BASE**::**BaseEPI** baseEPI,
in string contextName,
in string contextValue);

where the arguments identify the `BaseEPI` object that originated the interest, the changed variable name, and its value.

Only one such callback object can be registered. To unregister the callback object, call `unregisterContextInterest`.

Example

```
// callback client object implementation
void
HotContextObj::contextInterestCB(
              nt_EPI_BASE::BaseEPI_ptr objRef,
              const char * vname,
              const char * vval,
              CORBA::Environment & IT_env)
{
    /* react to the hot context selection
```

```
    whose component name value is in vval */
   ...
}
...
/* initialize the variable names string sequence */
nt_EPI_BASE::StringSeq vnames;
vnames.length(1);
vnames[0] = CORBA::string_dup("DPN_QUICK_STEP");
epiSvrRef->registerContextInterest(
              hotContextObjectRef,
              nt_EPI_BASE::BaseEPI::
                        CEPI_USER_CONTEXT,
              vnames);
```

API access

The API Access interfaces provide access to the MDM API communication mechanism. A generic access to all API functionality is provided (establishing connections to the API servers, sending API requests, receiving replies both synchronously and asynchronously through a callback objecct reference) as well as specialized interfaces and operations for the Alarm&Status, Network Model and the Host Group Directory Service. See the figure "API access interfaces" (page 518).

The API Access is provided through the following object interfaces:

- **APIServant (API Access servant)**
  This object represents the main interface to the CORBA EPI API servant. It provides the basic utilities already described (see "Base" on page 508) as well as a creator methods to construct new Generic and specialized API interface objects.
- **GenAPI (Generic API interface object)**
  GenAPI provides a generic interface to all API functionality. All its operations and data are inherited by all the other specialized API interface objects.
- **GMDRAPI (Alarm&Status (GMDR) API interface object)**
  This object is a specialization of the Generic API interface object that offers operations specific to the Alarm&Status API (such as simplified connector, alarm structure, creating alarm sieves, and receiving alarms).

- **NM (Network Model API interface object)**
  This object is a specialization of the Generic API interface object that offers operations specific to the Network Model API (currently, just a simplified connector).
- **HGDSAPI (Host Group Directory Service API interface object)**
  This object is a specialization of the Generic API interface object that offers operations specific to the HGDS service (such as simplified connector, host-group structure, host, group, children and parent requests, and host-group reply reception).

Figure 6
API access interfaces

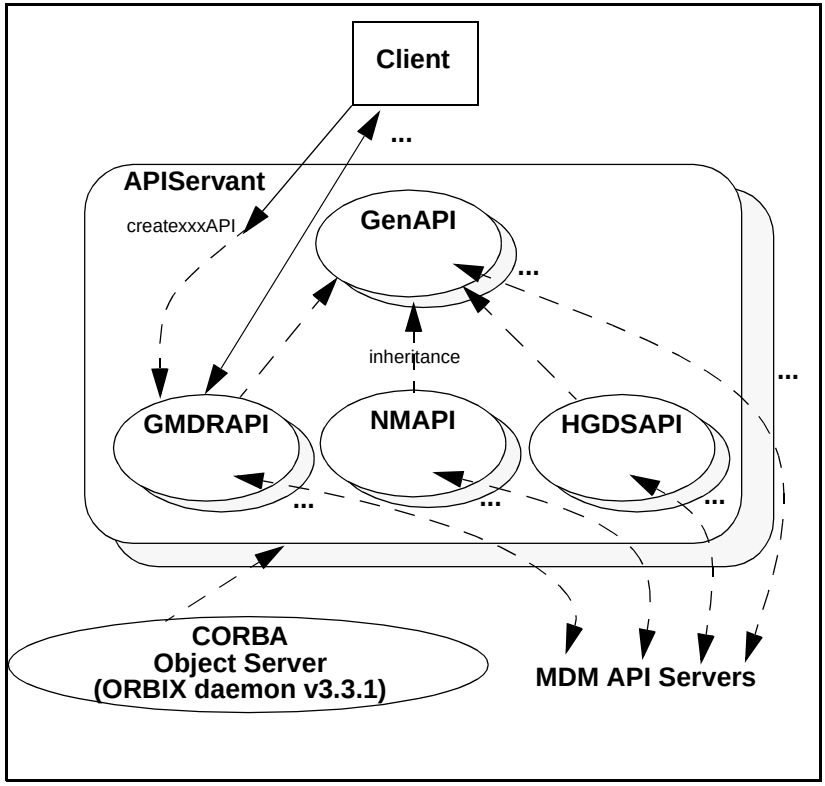
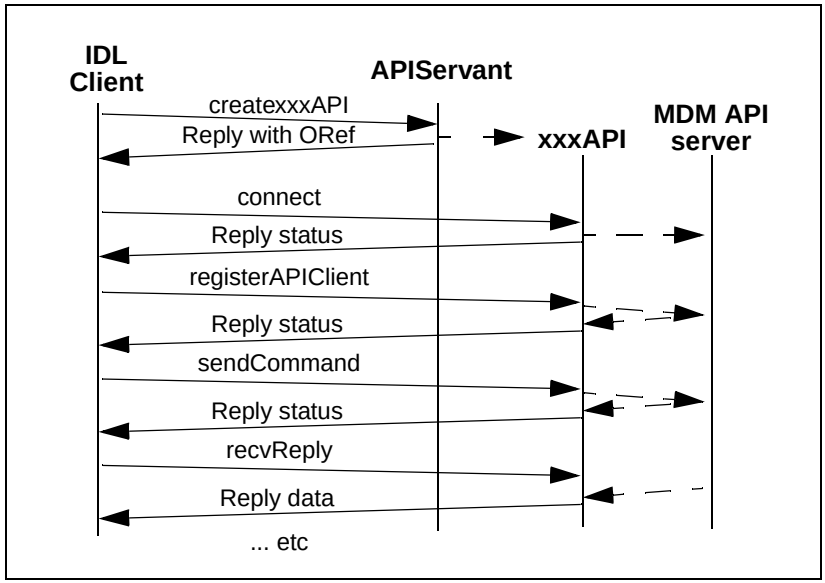


Figure 7
API access operations



The figure “API access operations” (page 519) provides a schematic of API access operations. API Access operations are used in the following sequence:

1 Use BIND or the Name Service interfaces to locate an API servant on the MDM server.

2 Through that object reference, request the creation of API interface objects.

3 Through the API object, connect to an appropriate MDM API server.

4 Register with that API server using the register operation.

5 Once registered, use that object to send API queries to the MDM server and retrieve its replies (synchronously). The specialized API object provides specialized operations for this. The generic API object provides general operations which are also usable from the specialized objects.

6 Alternatively, in asynchronous mode, create a local callback object and send a reference to it to the API Access interface as the object to callback with the reply data (one callback per interface). The Alarm&Status API object also supports a special means to do this to specify the way the replies (alarms) are returned.

7 When finished with an API object, request its termination through its `remove` operation.

8 Same for the API Access servant itself (which implies the destruction of all its remaining API objects.

Multiple API object can be created. An API object performs only one query at a time (note that it is possible to create a sieve through an API object yet still perform other API commands through it -- one at a time -- while the sieve is active and emitting notifications). The notifications may be interspersed along with the other command replies so care must be taken to properly interpret the responses.

API servant

Interface: `APIServant`
Module: `nt_API_ACCESS`
IDL file: `EPIAPI.idl`

To use the CORBA EPI API servant, you first have to register its implementation in the Orbix Implementation Registry (see “Enabling the CORBA EPI servants” on page 500).

The `APIServant` interface defines various types and constants used by all API interfaces as well as operations to create individual API interface objects.

When done with it, the API servant can be terminated (along with all its API interface objects) using its `remove` operation.

- **nt_API_ACCESS**::**GenAPI_*var*** api =
  nt_API_ACCESS::**APIServant**::**createGenAPI**(
  in string apiDictType)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Creates a new Generic API interface object and returns its object reference. One or more API interface (generic or specialized) can be

created. The `<apiDictType>` parameter indicates the type of API server that will be used from this interface. The possible values are defined in the `EPIAPI.idl` file in the `nt_API_ACCESS` module as:

`GENAPI_NM_DICT`	Network Model API
`GENAPI_GMDR_DICT`	Alarm & Status API
`GENAPI_HGDS_DICT`	Host Group Directory API
`GENAPI_IMDR_DICT`	Inbound Alarm & Status API
`GENAPI_NDAM_DICT`	Alarm & Status API, NDAM server

The generic API interfaces, as well as its own specialization, inherit and support all operations from the `BaseEPI` interface, see (“Common Operations” on page 509).

- **nt_API_ACCESS**::**GenAPI_*var*** api =
  nt_API_ACCESS::**APIServant**::**createGMDRAPI**()
  raises (**nt_CPI_EXCEPTION**::**CPIException**);

  nt_API_ACCESS::**GenAPI_*var*** api =
  nt_API_ACCESS::**APIServant**::**createAlarmStatusAPI**(
  in string apiDictType)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  These operations create a new Alarm&Status specialized API interface object and return its object reference. This interface offers additional operations specialized to its role. The first form is a shortcut to create a GMDR specific interface. The second form can be used to create an Alarm&Status API interface for another fault server like IMDR or NDAM, specifying the appropriate dictionary to use (see `nt_API_ACCESS::APIServant::createGenAPI`).
- **nt_API_ACCESS**::**NMAPI_*var*** api =
  nt_API_ACCESS::**APIServant**::**createNMAPI**()
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Does the same for a Network Model specialized API interface.

- **nt_API_ACCESS**::**HGDSAPI_*var*** api =
  nt_API_ACCESS::**APIServant**::**createHGDSAPI**()
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Does the same for a Host Group Directory Service specialized API interface.

  Example

```
// create an Alarm&Status API interface
// ... locate an API servant using BIND or the
//     naming service (episervRef)
try {
    nt_API_ACCESS::GMDRAPI_var apiRef =
        episervRef->createGMDRAPI();
    ...
```

- *void* **nt_API_ACCESS**::**APIServant**::**remove**();
  Terminates the API servant process and all its interface objects.

Generic API access

Interface: `GenAPI`
Module: `nt_API_ACCESS`
IDL file: `EPIAPI.idl`

The `GenAPI` interface inherits all operations from the `nt_EPI_BASE::CommonEPIOperations` interface. It provides for general API access operations and definition that can be used at that generic level or from a more specialized API Access interface.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GenAPI**::**connect**(
  in string servName,
  in string hostName)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  This is the general form of the `connect` operation to connect an API

Access interface to its providing server. Its first argument is a service name for that server. Typical values are defined within the `nt_API_ACCESS::GenAPI` interface as the following constant strings:

`CEPIGenAPI_NM_SERVICE`	Network Model API server
`CEPIGenAPI_GMDR_SERVICE`	Alarm & Status API server
`CEPIGenAPI_HGDS_SERVICE`	Host Group Directory API server
`CEPIGenAPI_IMDR_SERVICE`	Inbound Alarm & Status API server
`CEPIGenAPI_NDAM_SERVICE`	Alarm & Status and Network Model API, NDAM server
`CEPIGenAPI_GMDR_PREFIX`	set to "`GMDR_`"
`CEPIGenAPI_IMDR_PREFIX`	set to "`IMDR_`"
`CEPIGenAPI_NDAM_PREFIX`	set to "`NDAM_`"

The last three values are service name prefixes that can be used to construct alternate service names.

The `<hostName>` parameter indicates the workstation where the API service is expected to be running. It can either be a specific host name of one of the special Service Selection values below (also defined in the `nt_API_ACCESS::GenAPI` interface):

`CEPI_GMDR_SELECTED_HOST`	Surveillance Service Selected host
`CEPI_NM_SELECTED_HOST`	Network Model Service Selected host.
`CEPI_EM_SELECTED_HOST`	Passport Network Access Service Selected host

CEPI_DPN_SELECTED_HOST	DPN Network Access Service Selected host.
CEPI_DPNARCH_SELECTED_HOST	DPN Configuration Management Service Selected host.

Example

```
...
nt_API_ACCESS::GenAPI_var almapiref =
            epiServRef->createGenAPI(
                        GENAPI_GMDR_DICT);
if ( almapiref && !almapiref->is_nil() ) {
   if (almapiref->connect("GMDR", "localhost")
            == nt_EPI_BASE::CEPI_SUCCESS) {
   ... // connection successful
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_EPI_BASE::**CommonEPIOperations**::**isOK**();
  Returns current API server connection status of the interface (`nt_EPI_BASE::CEPI_SUCCESS`, `nt_EPI_BASE::CEPI_FAILED`, or `nt_EPI_BASE::CEPI_NO_CONNECTION`)

- **nt_EPI_BASE**::**EPI_Result**
  nt_API_ACCESS::**GenAPI**::**disconnect**()
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Disconnects the API Access interface from its current server. The interface can be reused by re-connecting it to the same ort another API server.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GenAPI**::**registerAPI**(
  in string userId,
  in string passwd,
  in string attr)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Sends a `REGISTER` to the interface with the specified parameters. If attrs is non-`NULL`, it provides a string containing additional attribute specifications in API syntax, one per line (for example, the "_attr: userCapability E mdInject" line needed to register to the IMDR server with alarm injection capabilities).

  This command is actually a full sequence (combination call) where the

REGISTER command is sent and its reply waited for. Consequently, the return code reflects on the registration's success, not only in sending the command

Example

```
...
    if (   (almapiref->connect("IMDR", "localhost")
             == nt_EPI_BASE::CEPI_SUCCESS)
        && (almapiref->registerAPI("ticketIf", "", "")
             == nt_EPI_BASE::CEPI_SUCCESS) ){
    ... // registration successful, ready to send
        // commands
```

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GenAPI**::**sendAPICommand**(
  in string command)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);

  Sends a query to the interface. The query is specified as an ASCII string in API syntax.

  Example

```
...
    if ( almapiref->sendAPICommand("_cmd: get\n\
_obj_type: network\n\
_obj_id: networkID S compRoot\n\
_scope: all\n\
_attr_id: all\n\
_filter:compID LEFT NI PM")
                == nt_EPI_BASE::CEPI_SUCCESS ) {
    ...
```

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GenAPI**::**recvAPIReply**(
  in long timeout,
  out **nt_API_ACCESS**::**GenAPI_RecordType**
  apiRecordType,
  out **nt_API_ACCESS**::**GenAPIFieldSeq** apiFieldSeq)
  raises (**nt_CPI_EXCEPTION**::**CPIException**)

  Waits for and receives the next reply record from the server. A timeout can be specified (by default, it waits forever -- nt_EPI_BASE::CEPI_TIMEOUT_FOREVER or -1). With a timeout of nt_EPI_BASE::CEPI_TIMEOUT_POLL, or 0, the command acts as a

no-wait poll. Other values wait for the specified number of seconds. The reply itself is returned as two out attributes; `<apiRecordType>` indicates the type of API record received. Its value is one of:

GENAPI_RT_NONE	Unknown record type
GENAPI_RT_REGISTER	REGISTER message reply
GENAPI_RT_RESPONSE	Response to a query
GENAPI_RT_EVENT	Event notification from a sieve
GENAPI_RT_ENDRESP	End of query or sieve creation
GENAPI_RT_ERROR	Error reply
GENAPI_RT_END	End of communication (dropping of API server connection)

`<apiFieldSeq>` is a CORBA sequence of `nt_API_ACCESS::GenAPIField` structures. Each element of the sequence represents one of the API lines (fields) in the response. The structure provides the following data elements:

nt_API_ACCESS:: **GenAPI_FieldType ftype**	is the API field type whose enumeration token values are; **GENAPI_FT_ATTR**, **GENAPI_FT_END_RESP**, **GENAPI_FT_ERROR**, **GENAPI_FT_END**, **GENAPI_FT_OBJ_TYPE**, **GENAPI_FT_OBJ_ID**, **GENAPI_FT_USER_ID**, **GENAPI_FT_CAPABILITY**, **GENAPI_FT_INV_ID**, **GENAPI_FT_EVENT_TYPE**, **GENAPI_FT_TIME**, and **GENAPI_FT_SIEVE_ID**.
string **name**	is the API field name element (for **_ATTR** and **_OBJ_ID** fields only).
string **type**	is the API field type element (for **_ATTR** and **_OBJ_ID** fields only).
string **value**	is the API field value.

Multiple-line attribute block values are returned as a single multiple-line value. Inapplicable fields may have *NULL* or "" values.

Example

```
if ( almapiref->sendAPICommand(...)
            == nt_EPI_BASE::CEPI_SUCCESS ) {
      ...
      nt_API_ACCESS::GenAPI_RecordType rtype;
      nt_API_ACCESS::GenAPIFieldSeq    * fields;
      while ( almapiref->recvAPIReply(
            nt_EPI_BASE::CEPI_TIMEOUT_FOREVER,
                        rType, fields)
            == nt_EPI_BASE::CEPI_SUCCESS ) {
            ... // process the reply
```

- *string* name =
  nt_API_ACESS::**GenAPI**::**getAPIRecordTypeName**(
  in **nt_API_ACCESS**::**GenAPI_RecordType**
  apiRecordType);
  This utility function simply provides a token to text mapping for the API Record Type enumeration values. The mapping used simply maps each Record Type enumeration token to its textual name.

- *string* name =
  nt_API_ACESS::**GenAPI**::**getAPIFieldTypeName**(
  in **nt_API_ACCESS**::**GenAPI_FieldType** apiFieldType);
  This utility function simply provides a token to text mapping for the API Field Type enumeration values. The mapping used simply maps each Field Type enumeration token to its textual name in lower-case characters and followed by a colon (for example, `_ATTR` maps to `"_attr:"`).

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GenAPI**::**skipRestOfReply**(
  out boolean error);
  This function waits for and ignores all further replies until the end of response. If it finds an error record in the replies the error argument is set to `True`.

- *void*
  nt_API_ACCESS::**GenAPI**::**registerCallback**(
  in **ClientAPIInterfaceCB** apiCBObj);
  void **nt_API_ACCESS**::**GenAPI**::**unregisterCallback**();
  Binds the specified callback client-side object reference `<apiCBObj>` to this Command Access interface object. This callback will be invoked whenever a new message is received from the server for this interface. Each API Access interface can bind a different callback object. The callback may be unbound with `unregisterCallback`. The callback object interface supports a oneway callback signature defined in the `APICB` interface (in the `EPIAPICallback.idl IDL`):

Interface: `ClientAPIInterfaceCB`
IDL file: `EPIAPICallback.idl`

oneway void
ClientAPIInterfaceCB::**notify**(
in **nt_API_ACCESS**::**GenAPI** apiIfRef,
in **nt_API_ACCESS**::**GenAPI_RecordType** apiRecordType,
in **nt_API_ACCESS**::**GenAPIFieldSeq** apiFieldSeq
in **nt_EPI_BASE**::**EPI_Result** status);

The parameters `<apiIfRef>` is a reference to the Command Access interface bound to the callback, `<apiRecordType>` is received API record type (as defined in the description of the `GenAPI::recvAPIReply` operation), `<apiFieldSeq>` are the received API record's fields (also described along with `GenAPI::recvAPIReply`), and `<status>` is one of `nt_EPI_BASE::CEPI_SUCCESS`, for a successful reception, `nt_EPI_BASE::CEPI_END` to indicate an end of response, `nt_EPI_BASE::CEPI_NO_CONNECTION`, if the connection to the server was lost, and `nt_EPI_BASE::CEPI_FAILED`, for other error cases.

Example

```
// callback client object implementation
void
APICBObj::notify(nt_API_ACCESS::GenAPI_ptr
                         apiIfRef,
                 nt_API_ACCESS::GenAPI_RecordType rType,
                 nt_API_ACCESS::GenAPIFieldSeq&
                         fields,
                 nt_API_BASE::EPI_Result status,
                 CORBA::Environment & IT_env)
{
    // ...Process the reply based on the status...
}
...
apiIfRef->sendAPICommand("_cmd: GET\n\
_obj_type: network\n\
_obj_id: networkId S compRoot\n\
_scope: next\n\
```

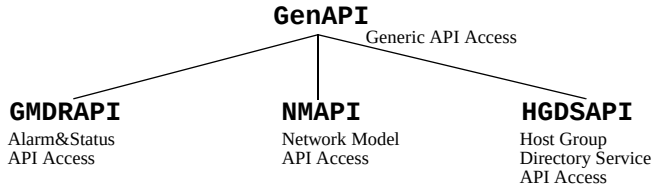
```
_attr_id: ALL\n");
apiIfRef->bindCallback(apiCBObjRef);
...
```

- *void*
  nt_API_ACCESS::**GenAPI**::**remove**();
  Destroys the API interface and frees all associated resources. Any future reference to the corresponding object from the CORBA client will fail.

Specialized API access

This section describes additional commands as well as value-added versions of the Generic API Access commands that are built to interact with specific APIs and their features.

The specialized API Access interfaces are derived from the GenAPI interface. All operations defined in the GenAPI interface are available to the API Access interfaces.



Alarm and Status API Access

Interface: `GMDRAPI`
Module: `nt_API_ACCESS`
IDL file: `EPIAPI.idl`

This interface adds operations and definitions to interact with the Alarm and Status API. In addition to a number of combination calls (for example, to create an alarm sieve or to inject alarms), it also supports automatic extraction of the major alarm attributes to support most forms of network automation.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GMDRAPI**::**connectOn**(
  in string hostName)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Connects to the GMDR server on the specified location (`<hostName>`). To connect the interface to an alternate Alarm&Status API server (IMDR, NDAM, alternate GMDR, ...) use the `GenAPI::connect` operation.

  Example

```
nt_API_ACCESS::GMDRAPIVar apiIfRef =
                 episervRef->createGMDRAPI();
if ( apiIfRef->connectOn(CEPI_GMDR_SELECTED_HOST)
                 == nt_EPI_BASE::CEPI_SUCCESS ) {
   // ... use the connected interface
```

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GMDRAPI**::**createAlarmSieve**(
  in boolean allflag,
  in string attrs,
  out long sieveId)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  This combination call creates a simplified alarm sieve and waits for the creation reply. The return code also indicates whether the creation was successful or not. By default, this sieve only extracts the major alarm attributes (`compId`, `time`, `severity`, `event`, `faultCode`, and `operatorData`) of all of the received alarms. However, if `<allflag>` is set to `True`, all of the attributes are extracted. You can add event filter (`"_attr: eventFilter SS <attribute> <operator> <type> <value>"`) and specific attribute extraction (`"_attr: eventInfo S <attribute>"`) parameters in API syntax through the `<attrs>` argument (with a carriage-return character -- `\n` -- between each line). Use an empty string `""` if none is needed.

  Each Alarm and Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned into the <sieveid> long integer out argument.

Example

```
// create an alarm sieve for all default attributes
// plus relatedComponents but only SET alarms being
// accepted
CORBA::Long sid;
if ( apiIfRef->createAlarmSieve(true,
    "_attr: eventFilter SS event EQ E SET\n"
    "_attr: eventInfo S relatedComponents\n", sid) {
    ... // sieve creation successful
```

The notifications from the sieve(s) can be collected with `GenAPI::recvAPIReply` or with `GMDRAPI::recvAlarm`.

Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GMDRAPI**::**recvAlarm**(
  in *long* timeout,
  in **nt_API_ACCESS**::**GMDRAPI**::**EPI_AlarmFormat** alarmFormat,
  out **nt_API_ACCESS**::**GenAPI_RecordType** apiRecordType,
  out **nt_API_ACCESS**::**GMDRAPI**::**GMDRAPI_Alarm** alarm,
  out string formattedAlarm,
  out **nt_API_ACCESS**::**GenAPIFieldSeq** apiFieldSeq)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);

  Enhances `GenAPI::recvAPIReply` by waiting for and receiving the next reply and automatically extracting the major alarm fields and formatting it if requested. As for `GenAPI::recvAPIReply`, `<timeout>` indicates how long the servant should wait for the next reply, `<apiRecordType>` and `<apiFieldSeq>` return the reply's API record type and fields. In addition, `<alarmFormat>` is used to request that the

interface also returns the alarm as a string preformatted to the standard MDM alarm formats. Its values (defined in the `GMDRAPI` interface) are:

`CEPI_ALARM_NONE_FORMAT`	No formatting is requested, the formattedAlarm is returned as an empty string (““).
`CEPI_ALARM_TERSE_FORMAT`	Alarm formatted as a one line summary
`CEPI_ALARM_NORMAL_FORMAT`	Like Terse but includes the Comment Data
`CEPI_ALARM_FULL_FORMAT`	All alarm information is provided.

`<alarm>` extracts the major alarm attributes and provides them as a handy structure defined in the `GMDRAPI` interface. Its fields are:

`string` **`compId`**	is the alarm's component ID
`string` **`time`**	is the alarm's time stamp in API format.
`nt_API_ACCESS::GMDRAPI::GMDRAPI_AlarmSeverityType severity`	is the alarm's severity value. The possible values are: **`warning`**, **`minor`**, **`major`**, **`critical`**, **`cleared`**, and **`indeterminate`**.
`nt_API_ACCESS::GMDRAPI::GMDRAPI_AlarmEventType event`	is the alarm's event type. Its possible values are: **`set`**, **`clear`**, and **`message`**.
`string` **`faultCode`**	is the alarm's fault code.
`string` **`operatorData`**	is the alarm's operator data.
`string` **`rawState`**	is the alarm's raw state.
`long` **`sieveId`**	is the sieve ID if the alarm is an event from a sieve (`0` otherwise).

Note: Some members may be returned as an empty `""` string if the corresponding field is not present. As well, `operatorData` can consist of more than one line. If the received message is not an alarm, the

formatted output is also returned as an empty string. For the formatted output to be correct, all attributes must have been requested by the `GET` request or the alarm sieve that is the source for the replied alarms.

A timeout can be specified. With a timeout of `nt_EPI_BASE::CEPI_TIMEOUT_POLL(0)`, the command acts as a no-wait poll. A value of `nt_EPI_BASE::CEPI_TIMEOUT_FOREVER(-1)` waits forever. Other values wait for the specified number of seconds.

For information on the legal values of these fields, see NTP 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GMDRAPI**::**recvStructuredAlarm**(
  in *long* timeout,
  out **nt_API_ACCESS**::**GenAPI_RecordType** apiRecordType,
  out **nt_API_ACCESS**::**GMDRAPI**::**GMDRAPI_Alarm** alarm,
  out **nt_API_ACCESS**::**GenAPIFieldSeq** apiFieldSeq)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Like `GMDRAPI::recvAlarm` but simplified not to provide alarms as a preformatted string.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GMDRAPI**::**recvFormatAlarm**(
  in *long* timeout,
  in **nt_API_ACCESS**::**GMDRAPI**::**EPI_AlarmFormat** alarmFormat,
  out **nt_API_ACCESS**::**GenAPI_RecordType** apiRecordType,
  out string formattedAlarm,
  out **nt_API_ACCESS**::**GenAPIFieldSeq** apiFieldSeq)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Like `GMDRAPI::recvAlarm` but simplified not to only provide alarms as as a special structure.

Example

```
CORBA::Long sid;
apiIfRef->createAlarmSieve(true,
                "_attr: eventFilter SS event EQ E SET",
```

```
                sid);
...
CORBA::String_var fAlarm;
nt_API_ACCESS::GenAPI_RecordType          rType;
nt_API_ACCESS::GenAPIFieldSeq             *fields;
nt_API_ACCESS::GMDRAPI::GMDRAPI_Alarm     *alarm;
if ( apiIfRef->recvAlarm(
                nt_EPI_BASE::CEPI_TIMEOUT_FOREVER,
                nt_API_ACCESS::GMDRAPI::
                          CEPI_ALARM_FULL_FORMAT,
                rType, alarm, fAlarm, fields)
          == nt_EPI_BASE::CEPI_SUCCESS ) {
     if (rType == nt_API_ACCESS::GENAPI_RT_EVENT) {
          if (alarm->severity ==
                nt_API_ACCESS::GMDRAPI::critical) {
                ... // handle the critical alarm
```

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GMDRAPI**::**injectAlarm**(
  in **nt_API_ACCESS**:**GMDRAPI**::
  GMDRAPI_AlarmEventType
  event,
  in **nt_API_ACCESS**:**GMDRAPI**::
  GMDRAPI_AlarmSeverityType
  severity,
  in string faultCode,
  in long notificationId,
  in string comment,
  in string time,
  in string attrs)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);

  Sends an alarm injection command with the following specified parameters:

  `compId` is the component ID.

  `event` is the alarm event (`set|clear|message`).

  `severity` is the severity of the alarm (`warning|minor|major|critical|cleared|indeterminate`).

faultCode is the fault code in AAAACCCC format.

notificationId is the sequence number (specified as an integer). If 0, a unique value will be provided.

comment is the comment data string.

time is the alarm time in API "YYYY MM DD hh mm ss" format, specify as an empty string *("")* to let IMDR assign the current time.

attrs can be used to assign additional attributes specified in API "_attr: <name> <type> <value>" line format (with \n as line separators).

Unspecified attributes take default values (see NTP 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*).

Note: This is not a combination call since the command does not wait for the server's reply, which must then be explicitly received or ignored.

Example

```
apiIfRef->injectAlarm("MYRTR WEST3",
                nt_API_ACCESS::GMDRAPI::set,
                nt_API_ACCESS::GMDRAPI::major,
                "A0000001", 23,
                "The router does not reply.",
                "_block: _attr operatorData S \
Try: 12, Timeout: 5, IP: 33.24.1.1 \
_end_block:");
CORBA::Boolean ignore;
apiIfRef->skipRestOfReply(ignore);
```

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GMDRAPI**::**createRawStateSieve**(
  in string attrs,
  out long sieveId)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  This combination call creates a simplified raw state change sieve and waits for the creation reply. The return code also indicates whether the

creation was successful or not. By default, this sieve only extracts the `compId`, `time`, and `rawState` attributes. It is possible to add an event filter (`"_attr: eventFilter SS <attribute> <operator> <type> <value>"`) with the `<attrs>` argument (with a carriage-return character -- `\n` -- between each line). Use an empty string if none is needed.

Each Alarm and Status API sieve is given a unique identification (the Sieve ID) which can be used in callbacks to identify the origin of the received event. The Sieve ID of the newly created sieve is returned into the long integer variable referred to by the `<sieveid>` argument.

Example

```
CORBA::Long sieveId;
if ( apiIfRef->createRawStateSieve(
          "_attr: eventFilter SS rawState EQ E OOS",
          sieveId) == nt_EPI_BASE::CEPI_SUCCESS ) {
   ... // sieve creation successful
```

The notifications from the sieve(s) can be collected with `GenAPI::recvAPIReply` or with `GMDRAPI::recvRawState`.

Note: Because the notifications are received asynchronously, they can be received while executing another API command. Be careful with combination calls that may ignore such notifications. It may be a better solution to use separate API interfaces for sieves and for commands.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**GMDRAPI**::**recvRawState**(
  in ***long*** timeout,
  out **nt_API_ACCESS**::**GenAPI_RecordType**
  apiRecordType,
  in **nt_API_ACCESS**::**GMDRAPI**::**GMDRAPI_RawState**
  rawState,
  out **nt_API_ACCESS**::**GenAPIFieldSeq** apiFieldSeq)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Enhances `GenAPI::RecvReply` by waiting for and receiving the next

reply and automatically extracting the raw state change fields. The extracted fields are returned as the `<rawstate>` structure defined in the IDL with the following fields:

string **compId**;	the target component ID.
string **time**;	is the notification time.
string **rawState**;	the raw state value.
long **sieveId**;	is sieve ID if this is an event from a sieve (`0` otherwise).

For information on the legal values of these fields, see NTP 241-6001-203 *Preside MDM Alarm and Status API Reference Guide*.

A timeout can be specified. With a timeout of `nt_EPI_BASE::CEPI_TIMEOUT_POLL` (`0`), the command acts as a no-wait poll. A value of `nt_EPI_BASE::CEPI_TIMEOUT_FOREVER` (`-1`) waits forever. Other values wait for the specified number of seconds.

Example

```
...
CORBA::Long sieveId;
apiIfRef->createRawStateSieve("", sieveId);
...
nt_API_ACCESS::GMDRAPI::GMDRAPI_RawState_var  rstate;
nt_API_ACCESS::GenAPI_RecordType              rType;
nt_API_ACCESS::GenAPIFieldSeq_var             fields;
if ( apiIfRef->recvRawState(
                nt_EPI_BASE::CEPI_TIMEOUT_FOREVER,
                rType, rstate.out(), fields.out())
        == nt_EPI_BASE::CEPI_SUCCESS ) {
   if (rType == nt_API_ACCESS::GENAPI_RT_EVENT) {
      if ( strcasecmp((const char*)rstate->rawState,
                      "OOS") == 0 ) {
        ... // handle the out-of-service component
```

- *void*
  nt_API_ACESS::**GMDRAPI**::**registerAlarmCallback**(
  in **ClientAPIInterfaceCB** apiCBObj,

in **nt_API_ACCESS**::**GMDRAPI**::**EPI_AlarmFormat**
format);

void
nt_API_ACESS::**GenAPI**::**unregisterCallback**();

Binds the specified callback client-side object reference `<apiCBObj>` to this Command Access interface object. This callback will be invoked whenever a new message is received from the server for this interface. Each API Access interface can bind a different callback object. The callback may be un-bound with `unregisterCallback`. The callback object interface supports a oneway callback signature defined in the `APICB` interface (in the `EPIAPICallback.idl IDL`):

Interface: `ClientAPIInterfaceCB`
IDL file: `EPIAPICallback.idl`

oneway void
ClientAPIInterfaceCB::**notifyAlarm***(*
in **nt_API_ACCESS**::**GMDRAPI** apiIfRef,
in **nt_API_ACCESS**::**GenAPI_RecordType**
apiRecordType,
in **nt_API_ACCESS**::**GMDRAPI**::
GMDRAPI_Alarm alarm,
in string formattedAlarm,
in **nt_API_ACCESS**::**GenAPIFieldSeq**
apiFieldSeq,
in **nt_EPI_BASE**::**EPI_Result** status*);*

The parameters `<apiIfRef>` is a reference to the Command Access interface bound to the callback, `<apiRecordType>` is received API record type (as defined in the description of the `GenAPI::recvAPIReply` operation), `<alarm>` is the Alarm key field structure as described for the `GMDRAPI::recvNextAlarm` operation, `<formattedAlarm>` is the preformatted alarm string as requested in the bind (an empty string if no formatting was requested), `<apiFieldSeq>` are the received API record's fields (also described along with `GenAPI::recvAPIReply`), and `<status>` is one of `nt_EPI_BASE::CEPI_SUCCESS`, for a successful reception,

`nt_EPI_BASE::CEPI_END` to indicate an end of response, `nt_EPI_BASE::CEPI_NO_CONNECTION`, if the connection to the server was lost, and `nt_EPI_BASE::CEPI_FAILED`, for other error cases.

Example

```
// callback client object implementation
void
APICBObj::notifyAlarm(nt_API_ACCESS::GenAPI_ptr
                      apiIfRef,
                nt_API_ACCESS::GenAPI_RecordType rType,
                nt_API_ACCESS::GMDRAPI::
                      GMDRAPI_Alarm&
                      alarm,
                const char * formattedAlarm,
                nt_API_ACCESS::GenAPIFieldSeq&
                      fields,
                nt_API_BASE::EPI_Result status,
                CORBA::Environment & IT_env)
{
   // ...Process the reply based on the status...
}
...
apiIfRef->sendAPICommand("_cmd: GET\n\
_obj_type: log\n\
_obj_id: logId S alarm\n\
_scope: next\n\
_attr_id: ALL\n");
apiIfRef->bindCallback(apiCBObjRef);
...
```

Network Model API Access

Interface: `NMAPI`

Module: `nt_API_ACCESS`

IDL file: `EPIAPI.idl`

The Network Model (NM) API access allows you to initialize an API interface to the server and connect the interface to the host.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**NMAPI**::**connectOn**(
  in string hostName)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Connects to the Network Model server on the specified location (<hostName>).

  Example

```
nt_API_ACCESS::NMAPI_var apiIfRef =
                epiServRef->createNMAPI();
if ( apiIfRef->connectOn(CEPI_NM_SELECTED_HOST)
                == nt_EPI_BASE::CEPI_SUCCESS ) {
   // ... use the connected interface
```

Various Generic API Access commands can be used to send queries and receive their replies.

Host Group Directory Service API Access

Interface: HGDSAPI
Module: nt_API_ACCESS
IDL file: EPIAPI.idl

The Host Group Directory Service (HGDS) API Access allows you to extract data on the Passport Group configuration of the Preside Multiservice Data Manager (MDM). Some value-added and combination calls are added to the Generic API Provider along with automatic extraction of the Passport Member information.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**HGDSAPI**::**connectOn**(
  in string hostName)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Connects to the Host Group Directory Service server on the specified location (<hostName>).

  Example

```
nt_API_ACCESS::HGDSAPI_var apiIfRef =
                epiServRef->createHGDSAPI();
if ( apiIfRef->connectOn(CEPI_EM_SELECTED_HOST)
                == nt_EPI_BASE::CEPI_SUCCESS ) {
   // ... use the connected interface
```

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**HGDSAPI**::**sendQuery**(
  in **nt_API_ACCESS**::**HGDSAPI**::**EPI_HGDS_SCOPE**
  scope,
  in string parameter)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Sends an HGDS API query as specified by the `<scope>` argument and optional `<parameter>`. The supported scopes are:

`CEPI_HGDS_GET_PARENT`	extracts the groups for the identified host name
`CEPI_HGDS_GET_CHILDREN`	extracts the member hosts for the identified group
`CEPI_HGDS_GET_GROUP`	extracts the specified group or all of them if `<parameter>` is an empty string
`CEPI_HGDS_GET_MEMBER`	extracts the specified member or all of them if `<parameter>` is an empty string

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_API_ACCESS::**HGDSAPI**::**recvHostGroup**(
  in ***long*** timeout,
  out **nt_API_ACCESS**::**GenAPI_RecordType**
  apiRecordType,
  out **nt_API_ACCESS**::**HGDSAPI**::
  EPI_HGDS_HostGroup
  hostGroup,
  out **nt_API_ACCESS**::**GenAPIFieldSeq** apiFieldSeq)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);

Enhances `GenAPI::recvAPIReply` to automatically extract the Passport host information if present. The extracted fields are as the `<hostgroup>` structure containing the following fields:

string **name**;	is the Passport host or group name.
string **ipAddr**;	is the Passport host IP address, if applicable., and
string **type**;	is "PASSPORT" for a host, and "GROUP" for a group.

A timeout can be specified. With a timeout of `nt_EPI_BASE::CEPI_TIMEOUT_POLL (0)`, the command acts as a no-wait poll. A value of `nt_EPI_BASE::CEPI_TIMEOUT_FOREVER (-1)` waits forever. Other values wait for the specified number of seconds.

Example

```
if ( apiIfRef->sendQuery(nt_API_ACCESS::HGDSAPI::
                         CEPI_HGDS_GET_CHILDREN,
                         groupName)
             == nt_EPI_BASE::CEPI_SUCCESS ) {
   nt_API_ACCESS::HGDSAPI::EPI_HGDS_HostGroup_var hg;
   nt_API_ACCESS::GenAPI_RecordType           rType;
   nt_API_ACCESS::GenAPIFieldSeq_var          fields;
   while ( apiIfRef->recvHostGroup(
             nt_EPI_BASE::CEPI_TIMEOUT_FOREVER,
                         rType, hg.out, fields.out)
             == nt_EPI_BASE::EPI_SUCCESS) {
    if (rType == nt_API_ACCESS::GENAPI_RT_RESPONSE) {
      sprintf(cmd, "/usr/sbin/ping %s",
             (const char*)(hg->ipAddress));
      if (system(cmd) == 0) {
             ... // Passport is reachable
```

Command access

Like the `cmccmd` utility, Command Access allows programs to connect to Passport Groups and DPN-100 OAs (the command route), send commands to the modules they contain, and receive the replies. Command Access also provides several utility commands to assist with the parsing and identification of the command output. Command Access is supported by a set of Command Session servers (CMCFUN and CM). See the figure "Command access interfaces" (page 545).

The CORBA EPI Command Access is provided through the following three Object interfaces:

- **CmdServant, the Command Access servant**
  This object represents the main interface to the CORBA EPI Command servant. It provides the basic utilities already described as well as a creator method to construct new Command Session objects.
- **CmdSession, the Command Session object**
  One or more Command Session object can be created on a servant (each will spawn its own set of Session Servers and lead to independent network connections). The main role of the command session (other than to control the life cycle of its command session, it to act as a factory of Command Interface objects.
- **CmdInterface, the Command Interface object**
  One or more Command Interface objects can be created from a Command Session object to mediate command flows to the network. Command Interfaces can process multiple commands but a given interface can only perform one command at a time (use multiple interfaces to issue commands in parallel -- which typically leads to the use of asynchronous mode for the reception of the replies).

Figure 8
Command access interfaces

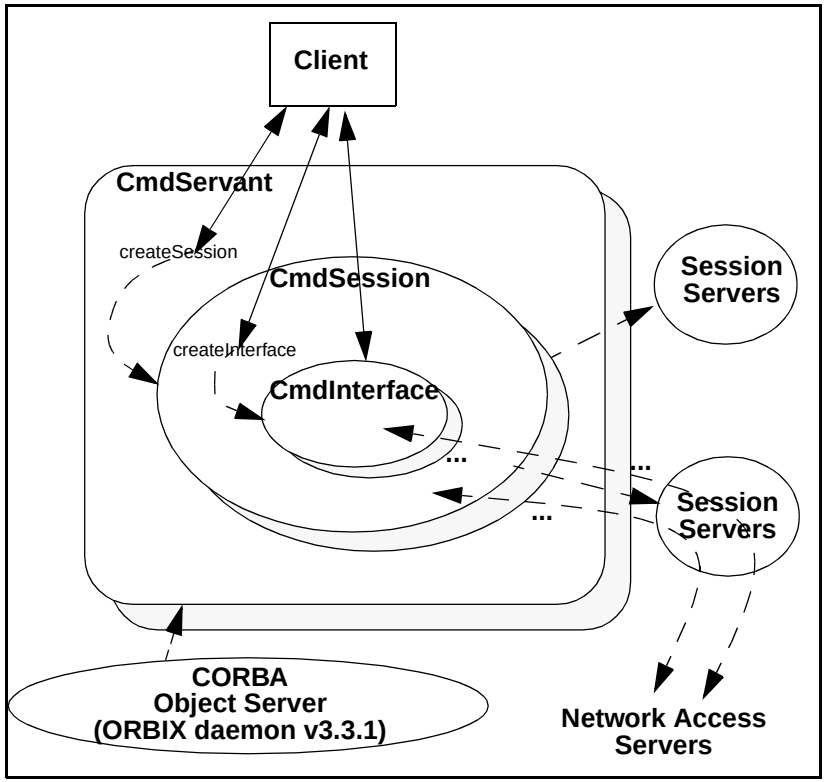
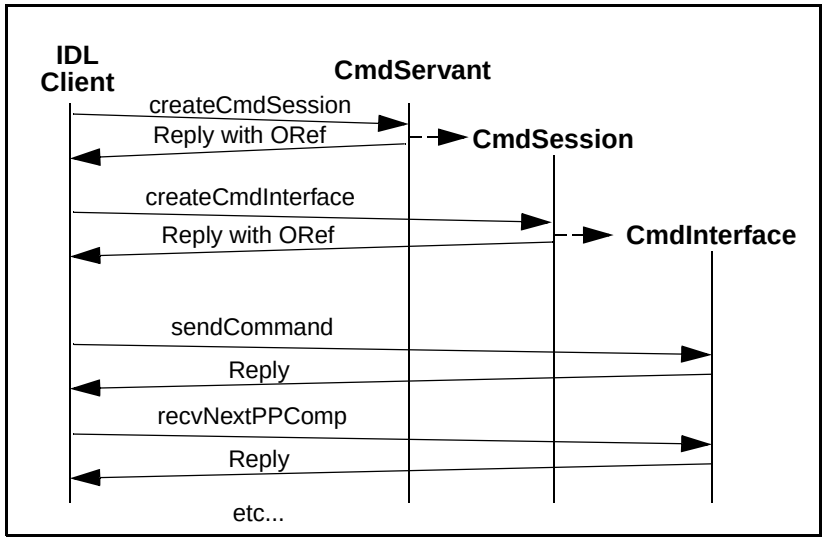


Figure 9
Command access operations



The figure “Command access operations” (page 546) provides a schematic of command access operations. Command Access operations are used in the following sequence:

1 Use BIND or the Name Service interfaces to locate a Command servant on the MDM server.

2 Through that object reference, request the creation of a Command Session object.

3 Through the Command Session object, request the creation of a Command Interface object.

4 Use that Command Interface object to issue commands and retrieve replies. This includes establishing group/OA connections, sending commands, retrieving reply data, and analyzing those replies. Use general utilities, if needed, provided by the servant object.

5 Alternatively, in Asynchronous mode, create a local callback object and send a reference to it to the Command Interface as the object to callback with the reply data (one callback per interface).

6 When finished with a Command Interface, request its termination through the its `remove` operation.

7 Same for Command Sessions (destroying a Command Session implicitly destroys all its associated Command Interfaces as well as the matching session servers).

8 Same for the Command servant itself (which implies the destruction of all its Command Sessions and their Command Interfaces.

Multiple commands can be sent to the nodes, but only one at a time per Command Interface.

Command servant

Interface: `CmdServant`
Module: `nt_CMD_ACCESS`
IDL file: `EPICmd.idl`

To use the CORBA EPI Command servant, you first have to register its implementation in the Orbix Implementation Registry. See “Enabling the CORBA EPI servants” on page 500. Note that the Command servant disables the use of the UNIX macro and SNMP command framework routes by default. To allow the use of these command routes remotely, you must have registered the servant interface with the enabled_mode option. See “Enabling the CORBA EPI servants” on page 500 for information on how to register the CORBA EPI servants.

The `CmdServant` interface defines operations to manage to manage `CmdSessions`. At least one `CmdSession` object must be created with the `createSession` call. Multiple such sessions can be created (each with a different name) which allows one to manage independent device connections. Two separate sessions may have connections to the same groups/OAs using different authentication levels or they can use the service of different Network Access MDM servers, both of which cannot be done using a single session. Typically, a single session will be sufficient though. Each session object implies the creation of a number of MDM processes on the servant side.

When done with, the command servant can be terminated (along with all its session and interface objects) using its `remove` operation.

- **nt_CMD_ACCESS**::**CmdSession_*var*** sess =
  nt_CMD_ACCESS::**CmdServant**::**createSession**(
  in string name,
  in string dpnHost,

in string ppHost)
raises (**nt_CPI_EXCEPTION**::**CPIException**);

Creates a private command session. The specified `<name>` must be a unique value for all sessions created by this servant. One or more session can be created. Each will spawn its own set of Command Session Servers and result in independent connections to the devices. The sessions will be automatically destroyed (along with all their command interfaces and connections) when the servant is terminated (notably if the client connection is lost).

If one of `<dpnAccessHost>` and/or `<ppAccessHost>` are not empty strings, the matching service selection is applied to the new session. The corresponding servers are used instead of the current workstation service selection.

Example

```
// create a command session
// ... locate a Command servant using BIND or the
// ... naming service (epiServRef)
try {
    nt_CMD_ACCESS::CmdSession_var cmdSessRef =
        epiServRef->createSession(
                "mysession", "", "");
    ...
```

The `CmdServant` interface inherits and supports all operations from the `BaseEPI` interface. See “Interface: BaseEPI Module: nt_EPI_BASE IDL file: EPIBase.idl” on page 509.

If the specified name is not unique, the `nt_CPI_EXCEPTION::CPI_SERVER_ERROR` exception is raised.

The session can be terminated through the `remove` operation available on from interface.

- *void*
  nt_CMD_ACCESS::**CmdServant**::**remove**();
  Terminates the command servant process and all its command sessions (and session servers) and interfaces. All device resources/connections are freed. Since the servant is terminated, no other commands can be sent.

Command Session

Interface: `CmdSession`
Module: `nt_CMD_ACCESS`
IDL file: `EPICmd.idl`

The `CmdSession` interface defines operations to manage `CmdInterfaces`. The actual interaction with the devices is performed though a `CmdInterface` object and each object handles one command at a time, synchronously (attempting to send a command through a `CmdInterface` while its command session is not completed is an error). To perform commands concurrently, if required, you need to create multiple `CmdInterface` objects with `createCmdInterface` (this usually implies the use of asynchronous operations using a client-side callback object). Each `CmdInterface` object implies a separate communication socket to the corresponding session servers. All `CmdInterface` objects share its group/OA connections with all the other `CmdInterfaces` of the same command session.

- **nt_CMD_ACCESS**::**CmdInterface_*var*** cmd =
  nt_CMD_ACCESS::**CmdSession**::**createCmdInterface**()
  raises (**nt_CPI_EXCEPTION**::**CPIException**);

  Creates a new command interface for the session. If the interface cannot be created, an exception is raised.

  Example

```
// create a command session
// ... locate a Command servant using BIND or the
// ... naming service (episervant)
try {
    nt_CMD_ACCESS::CmdSession_var cmdSessRef =
        epiSvrRef->createSession(
                "mysession", "", "");
    nt_CMD_ACCESS::CmdInterface_var cmdIfRef =
        cmdSessRef->createCmdInterface();
    ...
```

  If the interface cannot be created because of the lack of an active session, an `nt_CPI_EXCEPTION::EPIEX_NO_CONNECTION` exception is raised.

- **nt_EPI_BASE**::**EPI_Result** res =
  nt_CMD_ACCESS::**CmdSession**::**setServiceSelection**(
  in string dpnHost,
  in string ppHost)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Controls the Service Selection settings for the Command Session this interface is connected to. `<dpnHost>` specifies the MDM host name for DPN Network Access. `<ppHost>` specifies the MDM host name for Passport Network Access. If the value of these parameters is an empty string, then no change is made to the corresponding Service Selection. If other users (or programs/scripts) are using that session, then they will also be impacted by this change.

- *void* **nt_CMD_ACCESS**::**CmdSession**::**remove**();
  Terminates the current command session. All its command interfaces are automatically destroyed (their object reference will be invalid) as well as the associated MDM Command Session Servers. All device resources/ connections are freed.

Command Interface

Interface: `CmdInterface`
Module: `nt_CMD_ACCESS`
IDL file: `EPICmd.idl`

The `CmdInterface` interface defines operations to actually interact with the devices in the network. Operations are defined to connect to groups/OAs, send commands, retrieve replies in a variety of formats, manipulate those replies, and execute command templates.

- *void*
  nt_CMD_ACCESS::**CmdInterface**::**remove**();
  Destroys the command interface. Any group/OA connections made are retained in the session for other interfaces to use.

- **nt_EPI_BASE**::**EPIResult** res =
  isOK();
  Returns current session server connection status of the interface (`nt_EPI_BASE::CEPI_SUCCESS`, `nt_EPI_BASE::CEPI_FAILED`, or `nt_EPI_BASE::CEPI_NO_CONNECTION`)

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**sendConnect**(
  in **nt_CMD_ACCESS**::**CmdRouteType** routeType,
  in string routeName,
  in string userId,
  in string password)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);

  Sends a connect request to the indicated route using the specified authentication information. The reply from this message must be collected with `recvReply` (and the likes). The following route type enumeration values are defined for your convenience. Only the first two can be connected to, the others can be used in the other functions applicable to this interface type:

`CEPICMD_OA_ROUTE`	for DPN-100 NCS OAs (same as “`OA`”)
`CEPICMD_PP_GROUP_ROUTE`	for Passport groups (same as “`GROUP`”)
`CEPICMD_PP_WILD_ROUTE`	for the special Passport wild-card route (same as “`*`”)
`CEPICMD_MACRO_ROUTE`	for Unix Macros (same as “$”)[a]
`CEPICMD_SNMP_ROUTE`	for the SNMP Command framework (same as “@”)[a]

  a. The UNIX macro and SNMP Command routes are disallowed by the CORBA EPI Command servant by default. To allow their use, you must configure the servant as described in “Command servant” on page 547.

  If the connection request cannot be sent because of an already active command on the interface, an `nt_CPI_EXCEPTION::EPIEX_BUSY` exception is raised. If the connection to the session was lost, `nt_CPI_EXCEPTION::EPIEX_NO_CONNECTION` exception is raised instead.

Note: Connecting to an already connected route results in an error, even though the route is available. Send a "" command to the route to verify if a connection has already been established. The reply indicates an error if the connection does not exist.

Example

```
if (  (cmdIfRef->sendCommand("myOA", "")
              != nt_EPI_BASE::CEPI_SUCCESS)
    ||(cmdIfRef->skipRestOfReply(error)
              != nt_EPI_BASE::CEPI_SUCCESS) )
   if (cmdIfRef->sendConnect(CEPICMD_OA_ROUTE,
                        "myOA", "myCap", "aq1sw2")
              == nt_EPI_BASE::CEPI_SUCCESS) {
       CORBA::String_var txt;
       if (cmdIfRef->recvFullReply(txt)
              == nt_EPI_BASE::CEPI_SUCCESS) {
       ...    /* we're connected to the route */
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**sendDisconnect**(
  in string routeName)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Disconnects the interface from the named route (group or OA).

  If the disconnection request cannot be sent because of an already active command on the interface, an `nt_CPI_EXCEPTION::EPIEX_BUSY` exception is raised. If the connection to the session was lost, `nt_CPI_EXCEPTION::EPIEX_NO_CONNECTION` exception is raised instead.

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**sendCommand**(
  in string routeName,
  in string command)
  raises (**nt_CPI_EXCEPTION**::**CPIException**);
  Sends a command to a node through the specified connected route. The name of the destination node is typically the first token of the command.

  Example

```
if ( cmdIfRef->sendCommand("myOA", "R78 d")
                 == nt_EPI_BASE::CEPI_SUCCESS ) {
   ... /* command sent */
```

Note: The special command routes of the Command Console (“$” for macro access, “@” for the SNMP Command Framework, and “*” for the Passport wild-card group) can all be used as routes from EPI but they are normally disallowed for security reasons. To allow them, you must configure the CORBA EPI Command servant appropriately as described in “Command servant” on page 547.

If the command request cannot be sent because of an already active command on the interface, an `nt_CPI_EXCEPTION::EPIEX_BUSY` exception is raised. If the connection to the session was lost, `nt_CPI_EXCEPTION::EPIEX_NO_CONNECTION` exception is raised instead.

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**recvFullReply**(
  out string text)
  raises(**nt_CPI_EXCEPTION**::**CPIException**);
  Waits for and receives the complete reply to the previous node command. The replied text is returned as the `<text>` out parameter.

  Example

```
cmdIfRef->sendCommand(“myOA”, “dir”);
CORBA::String_var txt;
if (cmdIfRef->recvFullReply(txt)
                == nt_EPI_BASE::CEPI_SUCCESS) {
   if ( strstr((char*)txt, “pe”) != NULL ) {
...
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**recvNextLine**(
  out string text,
  in long timeout)
  raises(**nt_CPI_EXCEPTION**::**CPIException**);
  Waits for and receives the next line of the last reply of the issued node command. A time-out can be specified. With a time-out of `CEPI_TIMEOUT_POLL`, the command acts a no-wait poll. With `CEPI_TIMEOUT_FOREVER`, the Command servant blocks waiting for a reply. Other values wait for the specified number of seconds. The replied text is returned in `<text>`. The previous example can be rewritten as follows:

Example

```
cmdIfRef->sendCommand("myOA", "r78 d");
CORBA::String_var line;
while ( cmdIfRef->recvNextLine(line,
                nt_EPI_BASE::CEPI_TIMEOUT_FOREVER)
                == nt_EPI_BASE::CEPI_SUCCESS) {
   if ( strstr((char*)line, "pe") != NULL ) {
...
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**recvNextChunk**(
  out string text,
  in long timeout)
  raises(**nt_CPI_EXCEPTION**::**CPIException**);

  Waits for and receives the next chunk of the last reply of the issued node command. A chunk is most efficiently processed by EPI and may contain multiple lines of text (it may even end in the middle of a line). A timeout can be specified. With `timeout` set to `CEPI_TIMEOUT_POLL`, the command acts a no-wait poll. With `CEPI_TIMEOUT_FOREVER`, the command servant blocks waiting for a reply. Other values wait for the specified number of seconds. The replied text is returned in `<text>`. The previous example can be rewritten as follows:

  Example

```
cmdIfRef->sendCommand("myOA", "r78 d");
CORBA::String_var line;
while ( cmdIfRef->recvNextChunk(line
                nt_EPI_BASE::CEPI_TIMEOUT_FOREVER)
                == nt_EPI_BASE::CEPI_SUCCESS) {
   if ( strstr((char*)line, "pe") != NULL ) {
...
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**recvNextPPComp**(
  out **nt_CMD_ACCESS**::**CPIPPAttrSeq** attrs,
  out string componentId,
  in long timeout)
  raises(**nt_CPI_EXCEPTION**::**CPIException**);

  `recvNextPPComp` is a form of receive function that waits for and receives the next Passport component reply to the previous command (that is, the result of a Passport `list` or `display` command). The operation returns a string, `<componentId>`, with the received

component's name or an empty string, and `<attr>` the sequence of attribute value pairs for the received displayed attributes. `CPIPPAttrSeq` is a CORBA sequence of `CPIPPAttr` structures contains the following fields:

`string` **`name`**	the attribute name
`string` **`value`**	its value

If the Passport attribute is a list, vector or array, the first index is added to the name with a coma as separator (for example, `"pktFromIfByPrio,ep0"`). For two dimensional arrays, an entry is created with the attribute name and the column title as value. Another entry is created with "`<attribute name>,<row title>`" as a name and the list of column labels as value. The remaining entries for this attribute have "`<attribute name>,<row label>`" as name and the list of corresponding columns as values. Finally, the following special attributes are also available; `"Message" (name),` contains as value any message emitted by Passport not part of an attribute value (i.e. error messages), `"CompID" (name)` has for value the returned component's name.

A timeout can be specified. With timeout set to `CEPI_TIMEOUT_POLL`, the command acts as a no-wait poll. A value of `CEPI_TIMEOUT_FOREVER` waits forever. Other values wait for the specified number of seconds.

Note: Make sure you always specify the `-notab` (no tabular output) CAS command line option when sending a Passport CAS display command with wildcards if this command is to be used to extract the replies.

Example

```
cmdIfRef->sendCommand("myGroup", \
   "TOTO display shelf card/* utilization");
CORBA::String_var compId;
nt_CMD_ACCESS::CPIPPAttrSeq attrs;
while ( cmdIfRef->recvNextPPComp(attrs, compId,
```

```
                    nt_EPI_BASE::CEPI_TIMEOUT_FOREVER )
                    == nt_EPI_BASE::EPI_SUCCESS ) {
    cout << "Card: " << compid << endl;
...
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**skipRestOfReply**(
  out boolean error)
  This operation waits for and ignores all replies until the end of response. If a processing error is detected while receiving replies (not an on-switch error indication), then `<error>` will be set to `True`.

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**patternMatch**(
  in string pattern,
  out string outMatched);

  nt_EPI_BASE::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**patternSubstitute**(
  in string pattern,
  in string substitute,
  in int all,
  out string outSubstituted);
  This is the functionality of `nt_EPI_BASE::patternMatch` and `nt_EPI_BASE::patternSubstitute` applied to the last received command response (full, line, or chunk).

  Example

```
cmdIfRef->sendCommand("myOA", "dir");
CORBA::String_var line;
while ( cmdIfRef->recvNextLine(line,
                    nt_EPI_BASE::CEPI_TIMEOUT_FOREVER)
                    == nt_EPI_BASE::CEPI_SUCCESS ) {
    CORBA::String_var dontcare;
    if ( cmdIfRef->patternMatch(".*pe.*", dontcare)
                    == nt_EPI_BASE::CEPI_SUCCESS ) {
        cout << line << endl;
...
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**sendListRequest**(
  in string routeType);

  This special utility command requests a list of all OA, Passport Group, or both) types of available routes (when `<routeType>` set to `CEPICMD_OA_ROUTE` (“`OA`”), `CEPICMD_PP_GROUP_ROUTE` (“`GROUP`”), or `CEPICMD_ALL_ROUTE` (“`ALL`”) respectively). It corresponds to the `cmccmd list` MDM utility command and is similar in behavior to an `sendCommand` operation. The replies can be extracted with `recvNextLine` and have the following structure:

  ```
  <name> <type> <state>
  ```

  where `<name>` is the destination name, `<type>` is its type (`OA` or `GROUP`), and `<state>` its connection state (`CONN`, for connected, `AUTH`, for authentication, and `-` otherwise)

  Example

  ```
  cmdIfRef->sendDestRequest(“GROUP”);
  CORBA::String_var line;
  while (cmdIfRef->recvNextLine(line,
                nt_EPI_BASE::CEPI_TIMEOUT_FOREVER)
                == nt_EPI_BASE::EPI_SUCCESS) {
     if ( strstr(line, “CONN”) == 0 ) {
                ... route is available
  ```

  If the list request cannot be sent because of an already active command on the interface, an `nt_CPI_EXCEPTION::EPIEX_BUSY` exception is raised. If the connection to the session was lost, `nt_CPI_EXCEPTION::EPIEX_NO_CONNECTION` exception is raised instead.

- *void*
  nt_CMD_ACCESS::**CmdInterface**::**registerCallback**(
  in **ClientCmdInterfaceCB** clientCmdCBObj
  in **nt_CMD_ACCESS**::**RecvAsyncMode** mode);

  void
  nt_CMD_ACCESS::**CmdInterface**::**unregisterCallback**();
  `registerCallback` binds the specified callback client-side object reference `<clientCmdCBObj>` to this Command Interface object. This callback will be invoked whenever a new message is received from the server for this interface. Each Command Interface can bind a different

callback object. The <mode> parameter indicates how the received information is transferred to the object. If <mode> is set to CEPI_BY_LINE, the callback is invoked for every received line of text (similar to recvNextLine). If it is CEPI_BY_CHUNK, it is called for every chunk of text received (like recvNextChunk). Finally, if it is set to CEPI_BY_PPCOMP, the callback is invoked for every Passport Component description received (like recvNextPPComp). The callback may be un-bound with unregisterCallback. There are two oneway callback signatures defined in the ClientCmdInterfaceCB interface (in the EPICmdCallback.idl IDL):

Interface: ClientCmdInterfaceCB
IDL file: EPICmdCallback.idl

oneway void
ClientCmdInterfaceCB::**notify***(*
in **nt_CMD_ACCESS**::**CmdInterface** cmdIfRef,
in string text,
in **nt_EPI_BASE**::**EPI_Result** status*)*

called when the callback mode is CEPI_BY_LINE or CEPI_BY_CHUNK, and

oneway void
ClientCmdInterfaceCB::**notifyPPComp***(*
in **nt_CMD_ACCESS**::**CmdInterface** cmdIfRef,
in string componentId,
in **nt_CMD_ACCESS**::**CPIPPAttrSeq** attrs,
in **nt_EPI_BASE**::**EPI_Result** status*)*

called when the mode is CEPI_BY_PPCOMP.

The parameters are <cmdIfRef> is a reference to the Command Interface bound to the callback, <text> is the received text line or chunk (or an empty string if not applicable), <componentId> is the received Passport component's name or an empty string if not applicable, <attrs> is the sequence of Passport component attributes (see recvNextPPComp) received, and <status> is one of nt_EPI_BASE::CEPI_SUCCESS, for a successful reception,

`nt_EPI_BASE::CEPI_END` to indicate an end of response, `nt_EPI_BASE::CEPI_NO_CONNECTION`, if the connection to the server was lost, and `nt_EPI_BASE::CEPI_FAILED`, for other error cases.

Note that, contrary to the other EPI interfaces, the `patternMatch` and `patternSubstitute` operations cannot be used in the context of this callback to manipulate the received reply.

Example

```
/* callback client object implementation */
void
CmdCBObj::notify(
                nt_CMD_ACCESS::CmdInterface_ptr
                          cmdIfRef,
                const char * text,
                nt_EPI_BASE::EPI_Result status,
                CORBA::Environment & IT_env)
{
   // ...Process the reply text based on the status...
}
...
cmdIfRef->sendCommand("myOA", "r72 q serv");
cmdIfRef->registerCallback(cmdCBObjRef,
                nt_CMD_ACCESS::CEPI_BY_LINE);
...
```

- **nt_EPI_BASE**::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**doCommandFile**(
  in *string* filePath,
  in *string* dest,
  in **nt_CMD_ACESS**::**CPIAVLSeq** avl,
  out string output,
  out long result,
  in *boolean* bestEffort,
  in *boolean* trace);

  nt_EPI_BASE::**EPIResult** res =
  nt_CMD_ACCESS::**CmdInterface**::**doCommandFlow**(
  in *string* commandFlow,
  in *string* dest,

in **nt_CMD_ACESS**::**CPIAVLSeq** avl,
out string output,
out long result,
in *boolean* bestEffort,
in *boolean* trace);

These calls support the synchronous execution (commands and replies) of command flows from the file identified by `<filePath>` or the string identified by `<commandFlow>`. A command flow is a sequence of device commands (one per line) to be executed as one. The file is specified as a full path or a relative path searched for in `$HOME/MagellanNMS/<file path>`, then `/opt/MagellanNMS/cfg/<file path>`, and finally `/opt/MagellanNMS/lib/<file path>` on the Command servant's machine (`$HOME` refers to the home of the user the servant process is running as, most probably the root user).

All commands are executed through the named destination (if empty -- "" --, the destination must prefix every device command in the flow). The execution of the flow terminates at the first failure unless `<bestEffort>` is set to `True`. Variable substitution on each command in the flow is supported with the attribute-value pair list specified by `<avl>` as the source of the variable name-value pairs. `avl` is defined in the `EPICmd.idl` IDL as a `CPIAVLSeq` CORBA sequence of `CPIAVL` structures containing the following members:

string **name**	substitution variable name
string **value**	substitution variable value

The flow may produce some output text and numerical results which are returned as `<output>` and `<result>` respectively. The output text is produced from the flow using the `@PRINT` directive (see below). If `<trace>` is set to `True`, the executed device commands are also added to the output text (with a `#*` prefix for plain commands and `#?` for

conditional commands followed by their output and a `#? END`, `#? FAILED`, `#* END`, or `#* FAILED` line to indicate the end of the corresponding command).

The output <`result`> is produced from the flow by the `@EXIT` directive.

If the file/flow execution launched because of an already active command on the interface, an `nt_CPI_EXCEPTION::EPIEX_BUSY` exception is raised. If the connection to the session was lost, `nt_CPI_EXCEPTION::EPIEX_NO_CONNECTION` exception is raised instead.

Command flows consist of a list of device commands with, optionally, substitution variables identified by a dollar sign '`$`'. To specify a plain `$`, escape it as '`\$`'. For example, the following is a Passport command to set the <`committedInformationRate`> of a FrameRelay DLCI (read as one line):

```
$name set FrUni/$fruni Dlci/$dlci Sp\
            committedInformationRate $cir
```

A flow containing such a line could be executed with an attribute-value sequence containing at least: the following pairs

```
{“name”,  “NODER16”}
```

```
{“fruni”, “120”},
{“dlci”, “25”}
{“cir”,  “56000”}
```

Other forms of substitution variable specification include;

${<variable name>**}**	same as without the brackets
$!**<variable name>, or **${!<variable name>**}**	(strict substitution). When you use this variable in device commands, it does not apply to special directives. The command is skipped (silently not executed) if the specified variable has no associated value or is empty. This form is only available in actual device commands.
${<variable name>**:-** <default value>**}**	If the specified variable has no associated value or is empty, the specified default value substituted.
$%	This internal variable holds the contents of the last executed `@SWITCH` command or the value of the matched `\(\)` subpattern of the last executed `@case` command.
$%%	This special internal variable holds the pattern-matched contents (whole or sub-pattern) of the last executed `@IF` command (see below).

$?	This variable contains the numerical result of the last issued macro (@DO) or flow (@INCLUDE/@RUN).
${<variable name>**[**<index>**]}**	This represents an associative array value in the Flow language itself (not to be mistaken by array values in the scripting language). Such values can be directly set (@SET, @DEFINE, @LOCAL) or provided by specialized commands (@FOREACHPP, @SPLITCOMP, @SPLIT). When used, both the variable name and the index can be substituted variables (for example, ${array[$i]} in a loop that increments the value of $i). The ! and :- constructs apply also to the array entry (for example, ${array[$i]:-0} defining 0 as the default value for the entry).

Command flows also support special processing directives as indicated in the following table:

@PRINT <string>	append the specified text (after variable substitution) to the command output.
@FORMAT <multi-line string...> **@END**	same as @PRINT but for a multi-line piece of text.
@EXIT [<code>]	terminates the flow's execution with the specified result code. If no @EXIT directive is executed, the flow will have its result code set to 0 on success or 4 if the flow was terminated by a failed command.
@RETURN [<code>]	like @EXIT but when invoked from an included file (see @INCLUDE), only terminates the execution of that file, returning to the calling flow/file. If used from within a macro or included/run flow, the return value is available as the $? variable from the calling flow.
@TRY [<command>]	executes the command and ignores its possible failure (even if <bestEffort> was set to 0). If the command is omitted, sets the current operating mode for subsequent commands as if bestEffort had been specified as non-true. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.

@CRITICAL [<command>] or **@CRIT** [<command>]	executes the command and terminates the flow if the command fails even if <bestEffort> was set to a nonzero value. If the command is omitted, sets the current operating mode to critical for subsequent commands. Other modifiers (@TRACE/@NOTRACE, @LOG/@NOLOG) may also be specified.
@TRACE [<command>]	traces this command even if <trace> is set to 0. If the command is omitted, sets the current operating mode to trace (as if tracing had been enabled when the C/C++ call was made) for subsequent commands. Other modifiers (@TRY/@CRITICAL, @LOG/@NOLOG) may also be specified.
@NOTRACE [<command>]	does not trace this command even if -<trace> is set to a nonzero value. If the command is omitted, sets the current operating mode to no-trace for the subsequent commands. Other modifiers (@TRY/@CRITICAL, @LOG/@NOLOG) may also be specified.
@LOG [<command>]	logs this command as long as a log file has been defined (NMSCmdOpenCommandLog or @LOGFILE). If the command is omitted, sets the current operating mode to logging. Other modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE) may also be specified.

@NOLOG [<command>]	does not log this command, even if logging was enabled. If the command is omitted, sets the current operating mode to no-logging for the subsequent commands. Other modifiers (@TRY/ @CRITICAL, @TRACE/@NOTRACE) may also be specified.
@LOGFILE [[**+**]<log file path>]]	This controls command logging (see NMSCmdOpenCommandLog). Without arguments, this is equivalent to @LOG. If the log file is identified, the issued commands are logged to it from this point on (unless modified by @NOLOG). If the file path (on the servant's machine) is prefixed with the plus (+) sign, the logs are appended to the file if it exists (else the file is overwritten with the new logs).
@IF <var.> [**==**|**!=** <patterns>] or **@IF** <var.> **<**|**>**|**<=**|**>=** <value <commands> ... [**@ELSE** <commands> ...] **@END**	evaluates the specified variable expressions. This command executes the first command block if it finds that it matches one (==) or does not match any (!=) of the patterns in the first form. This command also compares (numeric or string, as appropriate), the second form, to the specified value. Otherwise, the command block following the @ELSE is executed, if there are any commands. If only the variable expression is specified, the test is positive if the expanded value is not empty. If the test was a pattern matching one, the value of the sub-pattern is available as the $%% internal variable.

```
@FOR <var> <from> <to>
       [<increment>]
or
@FOR <var> IN
       <token list>
<commands>
...
@END
or
@FOR <var> KEYS
       <array name>
<commands>
...
@END
```

executes the command block repeatedly. In the first form, the command block is executed repeatedly while incrementing the named (<var>) numerical variable in the AVL from <from> to <to> in jumps of <increment> (defaults to 1). In the second form, the command block is executed repeatedly while iterating the variable across the list of blank separated tokens. You can use the variable ($<var>) in the command block. You can terminate the loop prematurely by calling the @BREAK command from within the command block. Loops can be nested. The third form allows the variable to scan the existing indices of the named associative array (for example, the Passport attribute values from @FOREACHPP).

@FOREACHPP <var> <Passport list command> <commands> ... **@END**	executes the specified Passport list command then iterates the command over the returned component names, assigning its name to the named variable and executing the command block. You can terminate the loop prematurely by calling the @BREAK command from within the command block. Loops can be nested. If the Passport command was a display one, the extracted attribute values are also available as the entries of an associative array of the same name as the specified variable. They are provided in the same way as from the NMSCmdRecvNextPPComp function. If wild-cards are used, make sure you specify the -notab display command option.
@FOREACHLN <var> <command> <commands> ... **@END**	executes the specified command then iterates the command over the returned output line by line, assigning each one to the named variable and executing the command block. You can terminate the loop prematurely by calling the @BREAK command from within the command block. Loops can be nested.
@BREAK	invoked from within a @FOR/ @FOREACHPP/@FOREACHLN/@WHILE/ @WHILDO loop construct. This command terminates the enclosing loop prematurely. In other situations, the command acts like @RETURN with no return code.

@CONTINUE	invoked from within a @FOR/ @FOREACHPP/@FOREACHLN/@WHILE/ @WHILDO loop construct. This command causes the loop to immediately iterate to the next cycle. Like @BREAK, the following loop-code is not executed but unlike @BREAK, the loop is not abandoned.
@CB <text>	This command is not supported in templates used by the CORBA EPI interface.

```
@SWITCH <test command>
[<commands>]
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

executes the test command and executes the commands following the first `@CASE` whose patterns match the output of the test command (the optional commands that follow the `@SWITCH` before the first `@CASE` are always executed). Only one `@CASE` block in the `@SWITCH` construct is executed. `@SWITCH` blocks can be nested. Patterns are specified in GREP format with a '`|`' separating alternatives. If no pattern is specified, the `@CASE` block accepts any output.

The `@SWITCH` command is traced to output, if enabled, as:

```
#? <test command>
<command output...>
#? END
```

(If the command could not be executed, the last line will be **`#? FAILED`** instead).

The `@CASE` patterns may contain one sub expression (`\(\)`), each of whose matched value is available in the following code as the `$%` substitution variable. The usual modifiers (`@TRY`/`@CRITICAL`, `@TRACE`/`@NOTRACE`, `@LOG`/`@NOLOG`) can be specified after the `@SWITCH`.

```
@SWITCHVAL <value expr.>
@CASE [<patterns>]
<commands>
...
[@CASE [<patterns>]
<commands>
...]...
@END
```

This construct behaves much like `@SWITCH` but instead of getting its pattern match target value from a device command output, that value is directly specified as a parameter.

```
@WHILE <var.>
       [==|!=|<|>|<=|>=
       <patterns/
value>]
<commands>
...
@END
```

repeatedly executes the command block as long as the text (same as `@IF`) succeeds. The loop can be broken prematurely by invoking `@BREAK`.

```
@INCLUDE <file path>
or
@RUN <file path>
```

the named file (on the Command servant's machine) is included and processed as though its content was part of the current flow. If the file is identified as a relative path, the standard MDM search path applies (see above). The difference between `@INCLUDE` and `@RUN` is that variable definitions (`@DEFINE`) performed in the file invoked by `@INCLUDE` apply to the calling flow whereas those in a file invoked by `@RUN` are ignored upon return. If the file cannot be read, the flow's execution terminates unless `bestEffort` was set to a nonzero value or the `@TRY` prefix is used. The usual modifiers (`@TRY`/`@CRITICAL`, `@TRACE`/`@NOTRACE`, `@LOG`/`@NOLOG`) can be specified before `@INCLUDE`/`@RUN`.

```
@MACRO <name>
     [<parameters...>]
<code text>
...
@ENDMACRO
```

defines a new macro that can be executed later with `@DO/@IFDO/@WHILEDO`. The macro can be given a number of positional argument names to be provided when executed (the last specified parameter name gets all the remaining arguments passed). These parameter names have local scope to the macro (as if defined with `@LOCAL`). The macro can be recursive. Just like `@INCLUDE/@RUN`, it can be terminated by `@RETURN` which specifies a return code available as the `$?` variable to the caller. Variables defined/set by the macro have the same scope as the caller unless defined with `@LOCAL`. Macros must be defined before they are used. The macros themselves have global scope and can be redefined.

```
@DO <macro name>
      [<arguments...>]
```

invokes the named defined macro. The specified arguments are assigned (local scope to the macro) to the macro's parameter names -- the name gets the left over arguments). The `@RETURN`ed value from the macro is available as the `$?` internal variable.
The usual modifiers (`@TRY/@CRITICAL`, `@TRACE/@NOTRACE`, `@LOG/@NOLOG`) can be specified before `@DO`.

```
@IFDO <macro name>
     [<arguments...>]
<command>
...
[@ELSE
<commands>
...]
@END
```

merges the functionality of the @IF and @DO constructs. Executes the macro as for @DO then performs either the first command block if the macro returns a 0 result, else performs the second (@ELSE) command block if any. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @IFDO.

```
@WHILEDO <macro name>
     [<arguments...>]
<commands>
...
@END
```

merges the functionality of the @WHILE and @DO constructs. Repeatedly executes the macro as for @DO then the command block as long as the macro returns a 0 result. The usual modifiers (@TRY/@CRITICAL, @TRACE/@NOTRACE, @LOG/@NOLOG) can be specified after the @WHILEDO. As with @WHILE, the loop can be broken with @BREAK invoked in the command block.

```
@DEFINE <name> <value>
```

defines or redefines a variable name. The new value is available in the current command block and the blocks it invokes, notably for included files. For example, if a @DEFINE is invoked in a @CASE block, the modified value applies to the commands in that block but not in the commands that follow the @SWITCH construct for that @CASE. Similarly, @DEFINEs used in included files have no effect on the calling command block.

@UNDEFINE <name>	Contrary to @DEFINE, @SET, and @LOCAL, undefines the named variable. All matching associative array values are also undefined. To undefine a single entry in the array, specify its full name (for example, @UNDEFINE <array>[<index>]).
@LOCAL <name> [<value>]	Like @DEFINE but the new variable has local scope (it will not replace nor exist in the caller's scope, for example when used from within a macro or included flow). The local scope also applies to associative arrays by that name. This is useful when defining macros that need variables for which you do not want to override existing values.
@SET <name> <value1> [**+|-|*|/|%|~** <value2>]	similar to @DEFINE, but this command sets the variable to the result of the numerical expression (+, -, *, /, % -- remainder). The ~ operator performs a pattern match using the second value as a GREP style pattern list (| separated). The variable is set to the matching portion or to nothing. If the pattern contains a \(\) delineated sub-pattern, the matching sub-pattern is used as the new value. If only the name and first value is specified, this has the same effect as @DEFINE.

@SPLIT[**(**<separators>**)**]
<array name>
<string>

This creates a token of the specified string. The individual tokens are assigned to indexed elements of the named associative array (starting at 1). The actual variable's value is the number of resulting tokens. By default, tokens are done on blanks. Alternatively, the separator characters can be provided between brackets. For example, the following will print the individual applications in a Passport AVL, one per line:

```
@SPLIT(, \t\n) app $avl
@FOR i 1 $app
     @PRINT ${app[$i]}
@END
```

@SPLITCOMP <array name> <component ID>

analyzes the specified component ID and provides the results in the named associative array. The following (examples are based on the component EM/TOTO LP/2 Ds1/0.

- **$**<array> the component ID in API format (EM TOTO LP 2 DS1 0)
- **${**<array>**[_MOD]}** the module name (TOTO)
- **${**<array>**[_SUB]}** the subcomponent portion (minus first level) (LP/2 DS1/0)
- **${**<array>**[_PAR]}** the parent subcomp portion (minus last level) (EM/TOTO LP/2)
- **${**<array>**[**<category>**]}** the relative instance value for that level (EM -> TOTO, LP -> 2, DS1 -> 0)

@WAIT <nb seconds>

blocking wait for the specified number of seconds.

@ASK <name> [**:E**|**:I**|**:S**] [/validation patterns/] [=<default value>] <prompt string> **@CASK** <name> [**:E**|**:I**|**:S**] [/validation patterns/] [=<default value>] <prompt string>	These commands are not supported in templates used by the CORBA EPI interface.
# <comment>	comment line.
<device command>	actual device command, optionally with embedded variables ($ prefix) invoked after substitution. If the command fails, the flow execution terminates (<bestEffort> set to 0 or @TRY prefix). The command is traced to output, if enabled, as: **#*** <device command> <command output...> **#* END** (the last line will be **#* FAILED** instead if the command indicated an error).

Note: Note that @TRY, @CRITICAL, @TRACE, and @NOTRACE can be combined. They can also be used with the @INCLUDE directive. Also, the command specified with @SWITCH can also start with @TRACE or @NOTRACE.

Example

Assuming a file (examp.tmpl assumed to be in /opt/MagellanNMS/cfg) containing:

```
# Frame Relay QOS example
@SWITCH $name l FrUni/$fruni Dlci/$dlci
@CASE failed|ERROR
  @PRINT $name FrUni/$fruni Dlci/$dlci does not exist!
  @EXIT 1
@CASE
  $name set FrUni/$fruni Dlci/$dlci Sp cir $cir
  $name set FrUni/$fruni Dlci/$dlci Sp bc $bc
  $name set FrUni/$fruni Dlci/$dlci Sp be $be
  @IF $lmi
    $name set FrUni/$fruni Lmi procedures $lmi
    $name set FrUni/$fruni Lmi side $lmiside
  @END
@END
```

This flow can be executed from a CORBA EPI client with:

```
CORBA::Long result;
CORBA::String_var output;
nt_CMD_ACCESS::CPIAVLSeq avl;
avl.length(4);
avl[0].name = CORBA::string_dup("name");
avl[0].value = CORBA::string_dup("NODER16");
avl[1].name = CORBA::string_dup("fruni");
avl[1].value = CORBA::string_dup("120");
avl[2].name = CORBA::string_dup("dlci");
avl[2].value = CORBA::string_dup("25");
avl[3].name = CORBA::string_dup("cir");
avl[3].value = CORBA::string_dup("56000");
if ( cmdIfRef->doCommandFile("examp.tmpl", "*", avl,
                             output, result,
                             False, False)
            != nt_EPI_BASE::CEPI_SUCCESS ) {
   fprint("Failed (%d)!!!\n%s\n", result,
            (char*)output);
   exit(1)
}
```

If, instead, the contents of the file above is stored in a text buffer (for example, `flowString`), the flow can then be executed with:

```
CORBA::Long result;
CORBA::String_var output;
nt_CMD_ACESS::CPIAVLSeq avl;
```

```
avl.length(4);
avl[0].name = CORBA::string_dup("name");
avl[0].value = CORBA::string_dup("NODER16");
avl[1].name = CORBA::string_dup("fruni");
avl[1].value = CORBA::string_dup("120");
avl[2].name = CORBA::string_dup("dlci");
avl[2].value = CORBA::string_dup("25");
avl[3].name = CORBA::string_dup("cir");
avl[3].value = CORBA::string_dup("56000");
if ( cmdIfRef->doCommandFlow("*", flowString, avl,
                             output, result,
                             False, False)
                == nt_EPI_BASE::CEPI_SUCCESS ) {
    fprint("Failed (%d)!!!\n%s\n", result,
                (char*)output);
    exit(1)
}
```

Example

The following are small examples of flow code usage:

```
# determine the Passport version and save it
# for later use
@SWITCH $name d software avl
@CASE base_\([^ ,]*\)
  # $% contains the last \(\) match (the base version)
  @PRINT Passport version : $%
  @DEFINE ppversion $%
  # set variables accordingly (Tm subcomponent
  # introduced?)
  @IF $ppversion == CA.*|CB.*
    @DEFINE tm Tm
  @ELSE
    @DEFINE tm
  @END
@END
...
# use the variable defined above and set the
# p1 parameter to a default value if not set
$name set AtmIf/$atm Vcc/$vpci Vcd $tm txTdp 1 \
                ${p1:-64000}
...
```

```
# create multiple DS1 channels with @FOR
@FOR chan 0 24
  $name add Lp/$lp DS1/$ds1 Chan/$chan
  $name Lp/$lp DS1/$ds1 Chan/$chan timeslots $chan
  # run a secondary flow to create a FrUni
  # for each channel (the flow has access to
  # the current AVL including the $chan variable)
  @RUN addFrUni
@END
```

Test Mode

Test mode is not supported by the CORBA EPI Command Access interface. If you need to test your templates, use one of the scripting language EPIs to develop the template which can then be used (no test) by the CORBA EPI.

Sample CORBA EPI client programs

This section provides an example of use of the CORBA EPI interface from a C++ client.

Alarm Logger C++ EPI example

The following program uses the CORBA EPI API interface from C++ to log received alarms to file. More specifically, this example shows how to use the asynchronous callback capabilities from a single CORBA process. This example is available in the Preside Multiservice Data Manager (MDM) load in `/opt/MagellanNMS/cfg/macros/nms/src/AlarmLoggerCorba.cxx`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fstream.h>
#include <string.h>
#include <time.h>
#include <NamingService.hh>

// Include CORBA generated stubs
#include <EPIAPI.hxx>
#include <EPIAPICallback.hxx>
```

```
// implement the API Callback Interface object
// defined in the CORBA EPI IDLs
class APIClientCBImpl
    : public virtual ClientAPIInterfaceCBBOAImpl
{
public:
    APIClientCBImpl();
    virtual ~APIClientCBImpl();

    virtual void
    notify(nt_API_ACCESS::GenAPI_ptr api,
        nt_API_ACCESS::GenAPI_RecordType
                    apiRecordType,
        const nt_API_ACCESS::GenAPIFieldSeq&
                    apiFieldSeq,
        nt_EPI_BASE::EPI_Result status,
        CORBA::Environment &IT_env =
                    CORBA::IT_chooseDefaultEnv());

    virtual void
    notifyAlarm (nt_API_ACCESS::GMDRAPI_ptr gmdrapi,
        nt_API_ACCESS::GenAPI_RecordType rType,
        const nt_API_ACCESS::GMDRAPI::GMDRAPI_Alarm&
                    alarm,
        const char * formattedAlarm,
        const nt_API_ACCESS::GenAPIFieldSeq&
                    apiFieldSeq,
        nt_EPI_BASE::EPI_Result status,
        CORBA::Environment &IT_env =
                    CORBA::IT_chooseDefaultEnv());
};


#define APISERVERNAME  "0:ntApiAccessSvr"
#define FIRSTROOTNAME  (const char*)"ntEPI"
#define SECONDROOTNAME (const char*)"ntApiAccess"
#define SERVERNAME     (const char*)"ntApiAccessSvr"

CORBA::ORB_var              orb;
CORBA::Object_var           obj;
CosNaming::NamingContext_var rootContext;

nt_API_ACCESS::APIServant_var apiServantVar;
```

```
nt_API_ACCESS::GMDRAPI_var    gmdrAPIVar;

// log output control
int      count = 0;
ofstream outFile;
int      maxCount = 500;
char *   froot = "/opt/MagellanNMS/data/AlarmLog";
char *   hostName;
char *   alternate = NULL;


// Utilities to track errors
void
printCPIException(nt_CPI_EXCEPTION::CPIException& ex)
{
    switch (ex.exceptType) {
    case nt_CPI_EXCEPTION::EPIEX_NO_CONNECTION:
         cerr << "EPIEX_NO_CONNECTION";
         break;
    case nt_CPI_EXCEPTION::EPIEX_BUSY:
         cerr << "EPIEX_BUSY";
         break;
    case
    nt_CPI_EXCEPTION::CPI_CLT_CALLBACK_OBJ_NOT_VALID:
         cerr << "CPI_CLT_CALLBACK_OBJ_NOT_VALID";
         break;
    case  nt_CPI_EXCEPTION::CPI_SERVER_ERROR:
         cerr << "CPI_SERVER_ERROR";
         break;
    }

    cerr << " " << ex.exceptMessage << endl;
    exit(1);
}


void
printCORBAException(CORBA::SystemException& ex)
{
    cerr << "CORBA::SystemException" << ex << endl;
    exit(1);
}
```

```
void
printUnknownException()
{
    cerr << “unknown exception” << endl;
    exit(1);
}


// this function simply (re)opens a log file to
// spool further alarms
void
openOutFile(char * froot)
{
    char tname[16];
    char fname[256];
    time_t t = time(NULL);
    struct tm * ts = localtime(&t);

    if (outFile)
        outFile.close();

    // put a time-stamp in the spool file name
    strftime(tname, 15, “%y%m%d-%H%M%S”, ts);
    sprintf(fname, “%s%s.log”, froot, tname);

    outFile.open(fname);
    if (outFile == NULL)
        exit(1);
}


void
usage()
{
    cerr << “alarmlogger <bind <hostname> |
nameService> [<alternate> <maxCount> <fileroot>]”
         << endl;
    exit(1);
}
```

```
APIClientCBImpl::APIClientCBImpl()
{ // nothing needed for now
}


APIClientCBImpl::~APIClientCBImpl()
{ // nothing needed for now
}


void
APIClientCBImpl::notify(
        nt_API_ACCESS::GenAPI_ptr api,
        nt_API_ACCESS::GenAPI_RecordType rType,
        const nt_API_ACCESS::GenAPIFieldSeq&
                    apiFieldSeq,
        nt_EPI_BASE::EPI_Result status,
        CORBA::Environment &IT_env)
{ // not used in this program
}


// this callback method receives alarms from
// the API server. It logs them to a file.
void
APIClientCBImpl::notifyAlarm(
        nt_API_ACCESS::GMDRAPI_ptr gmdrapi,
        nt_API_ACCESS::GenAPI_RecordType rType,
        const nt_API_ACCESS::GMDRAPI::GMDRAPI_Alarm&
                    alarm,
        const char * formattedAlarm,
        const nt_API_ACCESS::GenAPIFieldSeq&
                    apiFieldSeq,
        nt_EPI_BASE::EPI_Result status,
        CORBA::Environment &IT_env)
{
   // if we reached the maximum for a file,
   // open up another one.
   if (++count > maxCount) {
        openOutFile(froot);
        count = 0;
   }

   // print the alarm preceeded by its key
```

```
    // fields and its length
    outFile << (alarm.compId
               ? (const char*)alarm.compId : "-")
            << endl;
    outFile << (alarm.time
                ? (const char*)alarm.time : "-")
            << endl;
    outFile << alarm.severity << endl;
    outFile << alarm.event << endl;
    outFile << (alarm.faultCode
                ? (const char*)alarm.faultCode : "-")
            << endl;
    outFile << strlen(formattedAlarm) << ' '
            << formattedAlarm << endl;
}

// binds to the CORBA EPI Fault Servant using
// Orbix bind
void
bind()
{
    try {
        apiServantVar =
            nt_API_ACCESS::APIServant::_bind(
                APISERVERNAME, hostName);
        if (CORBA::is_nil(apiServantVar)) {
            cerr << "Failed to bind to server" << endl;
            exit(1);
        }

        cout << "bind to " << APISERVERNAME
             << " on " << hostName << endl;
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }
}
```

```
// binds to the CORBA EPI Fault Servant using
// COS Naming Service
void
nameService()
{
    cout << "nameService() " << endl;

    // locate naming service object
    CORBA::Object_var namingService;
    CosNaming::NamingContext_var namingServiceLookup;

    // locate naming service
    try {
        namingService =
            CORBA::Orbix.resolve_initial_references(
                "NameService");
    } catch(...)  {
        cerr << "Exception thrown on
resolve_initial_reference of Naming Service"
            << endl;
        exit(1);;
    }

    if (CORBA::is_nil(namingService)) {
        cerr << "NamingService object is null" << endl;
        exit(1);
    }

    // attempt to narrow
    try {
        namingServiceLookup =
            CosNaming::NamingContext::_narrow(
                namingService);
    } catch(...)  {
        cerr << "Exception thrown on narrow to
CosNaming::NamingContext"
            << endl;
        exit(1);
    }

    if (CORBA::is_nil(namingServiceLookup)) {
        cerr << "CosNaming::NamingContext after narrow
is invalid"
```

```
                << endl;
            exit(1);
        }

        // locate the servant object
        CosNaming::Name_var name = new CosNaming::Name;

        name->length(3);
        name[0].id   = FIRSTROOTNAME;
        name[0].kind = (const char *)"";
        name[1].id   = SECONDROOTNAME;
        name[1].kind = (const char *)"";
        name[2].id   = SERVERNAME;
        name[2].kind = (const char *)"";

        try {
            CORBA::Object_var object =
            namingServiceLookup->resolve(name);

            if (!CORBA::is_nil(object)) {
                apiServantVar =
                    nt_API_ACCESS::APIServant::_narrow(
                        object);
                if (CORBA::is_nil(apiServantVar)) {
                    cerr << "After narrow servant object
is null"
                         << endl;
                    exit(1);
                }
            } else {
                cerr << "Null object returned from resolve
nt_API_ACCESS::apiServant"
                     << endl;
                exit(1);
            }
        } catch (...) {
            cerr << "Catching some exception on call to
resolve nt_API_ACCESS::apiServant"
                 << endl;
            exit(1);
        }
    }
```

```
// the main routine itself
int
main(int argc, char ** argv)
{
    // Initialize CORBA interface
    CORBA::ORB_var orb =
        CORBA::ORB_init(argc,argv,"Orbix");

    // extract hostname, alternate, the max and file
    // name root from the command line arguments
    if ( (argc < 2) || (argc > 6) ) {
        usage();
    }

    if (!strcasecmp(argv[1], "bind")) {
        if (argc == 6) {
            hostName  = argv[2];
            alternate = argv[3];
            maxCount  = atoi(argv[4]);
            froot     = argv[5];
        } else if (argc == 3) {
            hostName  = argv[2];
        } else {
            usage();
        }

        // use bind to get the CORBA object reference
        bind();
    } else if (!strcasecmp(argv[1], "nameService")) {
        if (argc == 5) {
            alternate = argv[2];
            maxCount  = atoi(argv[3]);
            froot     = argv[4];
        } else if (argc == 2) {
            // use default value for alternate,
            // maxCount and froot
        } else {
            usage();
        }

        // use NameService to get the CORBA object
        // reference
        nameService();
```

```
} else {
    usage();
}

// Initialize, Connect, and Register the
// Alarm&Status API Interface (service selected).
// (Dont' bother with error checking yet.)
if (CORBA::is_nil(apiServantVar)) {
    exit(1);
} else {
    try {
        gmdrAPIVar = apiServantVar
            ->createAlarmStatusAPI(
            nt_API_ACCESS::GENAPI_GMDR_DICT);

    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }
}

if (CORBA::is_nil(gmdrAPIVar)) {
    exit(1);
} else {
    try {
        gmdrAPIVar->connect(
nt_API_ACCESS::GenAPI::CEPIGenAPI_GMDR_SERVICE,
                hostName);
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }
}

try {
    gmdrAPIVar->registerAPI("AlarmLogger",
            NULL, NULL);
```

```
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }

    // create an alarm sieve and ask for all attributes
    nt_EPI_BASE::EPI_Result result;
    CORBA::Long             sieveId;

    try {
        result = gmdrAPIVar->createAlarmSieve(1, NULL,
                 sieveId);
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }

    if ( result != nt_EPI_BASE::CEPI_SUCCESS ) {
        // no sieve (no connection?), then nothing to
        // do.
        exit(1);
    }

    // open the first spooling file
    openOutFile(froot);

    // register this process as a servant (as it
    // supports the target callback object)
    try {

CORBA::Orbix.impl_is_ready("myAlarmLoggerSrv",
                0);  // set server name
    } catch (CORBA::SystemException& se) {
        cerr << "CORBA::SystemException" << endl
             << se << endl;
    } catch (...) {
        cerr << "Caught unknown exception while
```

```
launching server."
            << endl;
    }

    // instantiate the callback target object
    ClientAPIInterfaceCB_ptr apiCBPtr =
                new APIClientCBImpl();

    // register the target callback object with the
    // API interface (ask for alarms in full display
    // format)
    try {
        gmdrAPIVar->registerAlarmCallback(apiCBPtr,
    nt_API_ACCESS::GMDRAPI::CEPI_ALARM_FULL_FORMAT);
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }

    // loop forever processing notifications to the
    // target object
    try {

CORBA::Orbix.impl_is_ready("myAlarmLoggerSrv",
                CORBA::ORB::INFINITE_TIMEOUT);
    } catch (CORBA::SystemException& se) {
        cerr << "CORBA::SystemException" << endl
             << se << endl;
    } catch (...) {
        cerr << "Caught unknown exception while
launching server."
            << endl;
    }
}
```

Using the Sun C++ SparcCompiler (6.1/5.2) and the precompiled version of the IDL stubs, the program was compiled with the following command line:

```
/opt/SUNWspro/bin/CC -qoption ld -znodefs
      -features=no%conststrings \
      -library=iostream \
      -o AlarmLoggerCorba \
      -I/opt/MagellanNMS/lib/idl/cplusplus \
      -I$ORBIXHOME/include \
      /opt/MagellanNMS/cfg/macros/\
nms/src/AlarmLoggerCorba.cxx \
      -L/opt/MagellanNMS/lib -R /opt/MagellanNMS/lib \
        -lEPIPublic -lepistub \
      -L $ORBIXHOME/lib -R$ORBIXHOME/lib \
        -lorbix -lITtls -lITns \
      -L/usr/openwin/include/X11 \
      -R /usr/openwin/include/X11 \
        -lXt -lX11
```

You may need to recompile the stubs for your own use and language using the source IDLs found in `/opt/MagellanNMS/lib/idl/`.

Passport Card inventory C++ EPI example

The following program uses the CORBA EPI command interface from C++ to produce a card inventory of a specified Passport node. This example is available in the MDM load in `/opt/MagellanNMS/cfg/macros/nms/src/PPCardInvCorba.cxx`.

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <string>

// Include CORBA stuff
#include <NamingService.hh>

// Include CORBA EPI generated stubs
#include "EPIBase.hxx"
#include "EPICmd.hxx"
```

```
#define CMDSERVERNAME  "0:ntCmdAccessSvr"
#define FIRSTROOTNAME  (const char*)"ntEPI"
#define SECONDROOTNAME (const char*)"ntCmdAccess"
#define SERVERNAME     (const char*)"ntCmdAccessSvr"


CORBA::ORB_var               orb;
CORBA::Object_var            obj;
CosNaming::NamingContext_var rootContext;


nt_CMD_ACCESS::CmdServant_var cmdServantVar;
nt_CMD_ACCESS::CmdSession_var cmdSessionVar;
nt_CMD_ACCESS::CmdInterface_var cmdInterfaceVar;


char* hostName;
char* group;
char* mod;


// Utilties to track errors
void
printCPIException(nt_CPI_EXCEPTION::CPIException& ex)
{
   cerr << "nt_CMD_ACCESS::CPIException ";
   switch (ex.exceptType) {
   case nt_CPI_EXCEPTION::EPIEX_NO_CONNECTION:
        cerr << "EPIEX_NO_CONNECTION";
        break;
   case nt_CPI_EXCEPTION::EPIEX_BUSY:
        cerr << "EPIEX_BUSY";
        break;
   case
   nt_CPI_EXCEPTION::CPI_CLT_CALLBACK_OBJ_NOT_VALID:
        cerr << "CPI_CLT_CALLBACK_OBJ_NOT_VALID";
        break;
   case  nt_CPI_EXCEPTION::CPI_SERVER_ERROR:
        cerr << "CPI_SERVER_ERROR";
        break;
   }
   cerr << " " << ex.exceptMessage << endl;
   exit(1);
}
```

```
void
printCORBAException(CORBA::SystemException& ex)
{
    cerr << "CORBA::SystemException" << ex << endl;
    exit(1);
}


void
printUnknownException()
{
    cerr << "unknown exception" << endl;
    exit(1);
}


// binds to the CORBA EPI Command Servant using
// Orbix bind
void
bind()
{
    cout << "bind() " << hostName << endl;

    try {
        cmdServantVar =
            nt_CMD_ACCESS::CmdServant::_bind(
                    CMDSERVERNAME, hostName);
        if (CORBA::is_nil(cmdServantVar)) {

            cerr << "Failed to bind to server"
                 << endl;
            exit(1);
        }

        cout << "bind to " << CMDSERVERNAME
             << " on " << hostName << endl;
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
```

```
        printUnknownException();
    }
}


// binds to the CORBA EPI Command Servant using
// COS Naming Service
void
nameService()
{
    cout << "nameService() " << endl;

    // locate naming service object
    CORBA::Object_var namingService;
    CosNaming::NamingContext_var namingServiceLookup;

    // locate naming service
    try {
        namingService =
            CORBA::Orbix.resolve_initial_references(
                    "NameService");
    } catch(...) {
        cerr << "Exception thrown on
resolve_initial_reference of Naming Service"
             << endl;
        exit(1);
    }

    if (CORBA::is_nil(namingService)) {
        cerr << "NamingService object is null" << endl;
        exit(1);
    }

    // attempt to narrow
    try {
        namingServiceLookup =
            CosNaming::NamingContext::_narrow(
                    namingService);
    } catch(...) {
        cerr << "Exception thrown on narrow to
CosNaming::NamingContext"
             << endl;
        exit(1);
    }
```

```
    if (CORBA::is_nil(namingServiceLookup)) {

        cerr << "CosNaming::NamingContext after narrow
is invalid"
             << endl;
        exit(1);
    }

    // locate the servant object
    CosNaming::Name_var name = new CosNaming::Name;
    name->length(3);
    name[0].id   = FIRSTROOTNAME;
    name[0].kind = (const char *)"";
    name[1].id   = SECONDROOTNAME;
    name[1].kind = (const char *)"";
    name[2].id   = SERVERNAME;
    name[2].kind  = (const char *)"";

    try {
        CORBA::Object_var object =
            namingServiceLookup->resolve(name);
        if (!CORBA::is_nil(object)) {
            cmdServantVar =
                nt_CMD_ACCESS::CmdServant::_narrow(
                    object);
            if (CORBA::is_nil(cmdServantVar)) {
                cerr << "After narrow servant object
is null"
                     << endl;
                exit(1);
            }
        } else {
            cerr << "Null object returned from resolve
nt_CMD_ACCESS::CmdServant"
                 << endl;
            exit(1);
        }
    } catch (...) {
        cerr << "Catching some exception on call to
resolve nt_CMD_ACCESS::CmdServant"
             << endl;
        exit(1);
    }
}
```

```
// main function
int
main(int argc, char ** argv) {
    // Initialize CORBA interface
    CORBA::ORB_var orb;
    CORBA::ORB_init(argc,argv,"Orbix");

    // Parse arguments
    if ( (argc < 4) || (argc > 5) ) {
        cerr << "ppcardrepclient bind
<hostname>|nameService    <group> <passport>"
             << endl;
        exit(1);
    }

    if (!strcmp(argv[1], "bind") && (argc == 5)) {
        hostName = argv[2];
        group    = argv[3];
        mod       = argv[4];

        // use bind to get the CORBA object reference
        bind();
    } else if    (!strcmp(argv[1], "nameService")
              && (argc == 4)) {
        group = argv[2];
        mod   = argv[3];
        // use NameService to get the CORBA
        // object reference
        nameService();
    } else {
        cerr << "ppcardrepclient bind
<hostname>|nameService <group> <passport>"
             << endl;
        exit(1);
    }

    // create a session
    try {
        cmdSessionVar = cmdServantVar->createSession(
            "mySession", "", "");
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex)
```

```
            printCORBAException(ex);
        } catch (...) {
            printUnknownException();
        }

        // create a command interface
        try {
            cmdInterfaceVar =
                cmdSessionVar->createCmdInterface();
        } catch (nt_CPI_EXCEPTION::CPIException& ex) {
            printCPIException(ex);
        } catch (CORBA::SystemException& ex) {
            printCORBAException(ex);
        } catch (...) {
            printUnknownException();
        }

        // sendConnect to the node
        try {
            if ( cmdInteraceVar->sendConnect(
                nt_CMD_ACCESS::CEPICMD_PP_GROUP_ROUTE,
                group, "system", "robocop") !=
                        nt_EPI_BASE::CEPI_SUCCESS ) {
                cerr << "SendConnect failed " << endl;
                exit(1);
            }
        } catch (nt_CPI_EXCEPTION::CPIException& ex) {
            printCPIException(ex);
        } catch (CORBA::SystemException& ex) {
            printCORBAException(ex);
        } catch (...) {
            printUnknownException();
        }

        // get sendConnect result
        CORBA::String_var output;

        try {
            if ( cmdInterfaceVar->recvFullReply(output)
                        !=  nt_EPI_BASE::CEPI_SUCCESS )
            {
                cerr << "\n*** Failed to connect to the
node"
                        << endl;
```

```
            exit(1);
        }
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }

    // print the title
    cout << "\n Passport Card Inventory\n"
         << "----------------------------------------
--\
-----------------------------\n"
         << "Node         Card Type         \
Inserted     Serial #       Firm. Rev.     LP\n"
         << "------------ ---- ------------ \

------------ ------------- ------------- ------"
         << endl;

    // Get the number of slots from the shelf component.
    char cmdline[256];
    int  numberOfSlots = 16;
    sprintf(cmdline, "%s d shelf numberOfSlots", mod);

    try {
        if ( cmdInterfaceVar->sendCommand(group,
                cmdline) !=
                    nt_EPI_BASE::CEPI_SUCCESS ) {
            cerr << "SendCommand (nbslots) failed"
                 << endl;
            exit(1);
        }
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }
```

```
nt_CMD_ACESS::CPIAVLSeq* avl = NULL;
CORBA::String_var        compId;
CORBA::Long          forever =
        nt_EPI_BASE::CEPI_TIMEOUT_FOREVER;
CORBA::Boolean             skip;

try {
    if ( cmdInterfaceVar->recvNextPPComp(avl,
                compId, forever)
            == nt_EPI_BASE::CEPI_SUCCESS ) {
        int i;
        for ( i = 0; i < avl->length(); i++ ) {
            if (strcasecmp((*avl)[i].name,
                    "numberOfSlots") == 0) {
                numberOfSlots =
                    atoi((*avl)[i].value);
                break;
            }
        }
    }
    delete avl;
    cout << "numberOfSlots is " << numberOfSlots
         << endl;
    cmdInterfaceVar->skipRestOfReply(skip);
} catch (nt_CPI_EXCEPTION::CPIException& ex) {
    printCPIException(ex);
} catch (CORBA::SystemException& ex) {
    printCORBAException(ex);
} catch (...) {
    printUnknownException();
}

// Get the needed attributes from all card
// components.
char card[64];
char cardType[64];
char insertedCardType[64];
char serialNumber[64];
char activeFirmwareVersion[64];
char currentLp[64];
cardType[0]         = '\0';
insertedCardType[0] = '\0';
```

```
    sprintf(cmdline, "%s d -notab shelf card/*
cardType,insertedCardType,serialNumber,activeFirmware
Version,currentLP", mod);

    try {
        if ( cmdInterfaceVar->sendCommand(group,
                cmdline) !=
                    nt_EPI_BASE::CEPI_SUCCESS ) {
            cerr << "SendCommand c (card info) failed"
                 << endl;
            exit(1);
        }
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }

    try {
        while ( cmdInterfaceVar->recvNextPPComp(avl,
                    compId, forever)
                == nt_EPI_BASE::CEPI_SUCCESS ) {
            // If this is the name of a card (either
            // the first or another one in the list).
            int cardnb;
            if ( sscanf(compId, "Shelf Card/%d",
                    &cardnb) == 1 ) {
                if (   ( cardType[0] != '\0' )
                    && ( insertedCardType[0]
                         != '\0' )
                    && (   (strcasecmp(cardType,
                                "none") != 0)
                        || (strcasecmp(
                              insertedCardType,
                                "none") != 0) ) ) {
                    // Print the info on the
                    // preceding card.
                    cout << setiosflags(ios::left)
                         << setw(12) << mod
                         << setw(4)  << card
                         << setw(12) << cardType
```

```
                        << setw(12)
                        << insertedCardType
                        << setw(14)
                        << serialNumber
                        << setw(14)
                        << activeFirmwareVersion
                        << setw(0)  << currentLp
                        << endl;
                    cardType[0]         = '\0';
                    insertedCardType[0] = '\0';
                }
                // stop if we've reached the maximum
                if ( cardnb > numberOfSlots )
                    break;
            }

            // extract the card attributes
            int i;
            for ( i = 0; i < avl->length(); i++ ) {
                if      ( strcasecmp((*avl)[i].name,
                          "cardType") == 0 )
                    strcpy(cardType,
                          (*avl)[i].value);
                else if ( strcasecmp((*avl)[i].name,
                          "insertedCardType") == 0 )
                    strcpy(insertedCardType,
                          (*avl)[i].value);
                else if ( strcasecmp((*avl)[i].name,
                          "activeFirmwareVersion")
                    == 0 )
                    strcpy(activeFirmwareVersion,
                          (*avl)[i].value);
                else if ( strcasecmp((*avl)[i].name,
                          "serialNumber") == 0 )
                    strcpy(serialNumber,
                          (*avl)[i].value);
                else if ( strcasecmp((*avl)[i].name,
                          "currentLp") == 0 )
                    strcpy(currentLp,
                          (*avl)[i].value);
            }
        } // while
```

```
        // Print the last card's information if
        // required.
        if (   ( cardType[0]         != '\0' )
            && ( insertedCardType[0] != '\0' )
            && (   (strcasecmp(cardType,
                          "none") != 0)
                || (strcasecmp(insertedCardType,
                          "none") != 0) ) ) {
            cout << setiosflags(ios::left)
                 << setw(12) << mod
                 << setw(4)  << card
                 << setw(12) << cardType
                 << setw(12) << insertedCardType
                 << setw(14) << serialNumber
                 << setw(14) << activeFirmwareVersion
                 << setw(0)  << currentLp
                 << endl;
        }
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }

    // sendDisconnect
    try {
        if ( cmdInterfaceVar
                 ->sendDisconnect(group)
                     != nt_EPI_BASE::CEPI_SUCCESS ) {
            cerr << "SendDisconnect failed "
                 << endl;
            exit(1);
        }
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }
```

```
    try {
        if ( cmdInterfaceVar
                ->recvFullReply(output)
                    != nt_EPI_BASE::CEPI_SUCCESS ) {
            cerr << "\n*** Failed to disconnect to the
node"
                 << endl;
            exit(1);
        }
    } catch (nt_CPI_EXCEPTION::CPIException& ex) {
        printCPIException(ex);
    } catch (CORBA::SystemException& ex) {
        printCORBAException(ex);
    } catch (...) {
        printUnknownException();
    }

    // clean up
    try {
        cmdInterfaceVar->remove();
        cmdSessionVar->remove();
        cmdServantVar->remove();
    } catch (CORBA::SystemException& ex) {
        printCPIException(ex);
    } catch (...) {
        printUnknownException();
    }

    exit(0);
}
```

Using the Sun C++ SparcCompiler (6.1/5.2) and the precompiled version of the IDL stubs, the program was compiled with the following command line:

```
/opt/SUNWspro/bin/CC -qoption ld -znodefs
      -features=no%conststrings \
      -library=iostream \
      -o PPCardInvCorba \
      -I/opt/MagellanNMS/lib/idl/cplusplus \
      -I$ORBIXHOME/include \
      /opt/MagellanNMS/cfg/macros/\
nms/src/PPCardInvCorba.cxx \
```

```
-L/opt/MagellanNMS/lib -R /opt/MagellanNMS/lib \
  -lEPIPublic -lepistub \
-L $ORBIXHOME/lib -R$ORBIXHOME/lib \
  -lorbix -lITtls -lITns \
-L/usr/openwin/include/X11 \
-R /usr/openwin/include/X11 \
  -lXt -lX11
```

You may need to recompile the stubs for your own use and language using the source IDLs found in `/opt/MagellanNMS/lib/idl`.

Appendix A
DoEPITemplate Utility

This appendix describes the **DoEPITemplate** utility.

DoEPITemplate simplifies the execution of EPI Command interface command flows (templates). **DoEPITemplate** may be used to execute the following templates:

- operational for DPN and Passport
- configurational for Passport

For configurational templates, the **DoEPITemplate** also automates and simplifies the application of configuration templates. The **DoEPITemplate** also controls the provisioning session for the templates. Multiple templates can be applied at the same time. The templates (and provisioning session) can be applied to one or more nodes to support self-similar configuration, multi-node configuration, or correlated configuration. For example, adding a user to all Passport nodes, or end-to-end DLCI configuration.

DoEPITemplate may be run against the following sessions:

- current user/command
- specific, already existing session
- specially created

DoEPITemplate may automatically trigger the backup of the modified configuration and the population of the NRS database at the end of the configuration process.

The utility can output a detailed log of its operations if you use the **-trace** option. The utility also outputs the names of nodes that have been processed successfully to the file /tmp/DoEPITemplate.failednodes. The file contains one node name per line, with the final configuration file name for configurational mode operations. The failed nodes are contained in the file /tmp/DoEPITemplate.successnodes. The file name prefix may be changed with the **-log** option).

Command line arguments

The **DoEPITemplate** utility can be invoked as /opt/MagellanNMS/bin/DoEPITemplate. The figure “Command line options” (page 609) and the table “DoEPITemplate command options” (page 610) describe the command line options.

Figure 10
Command line options

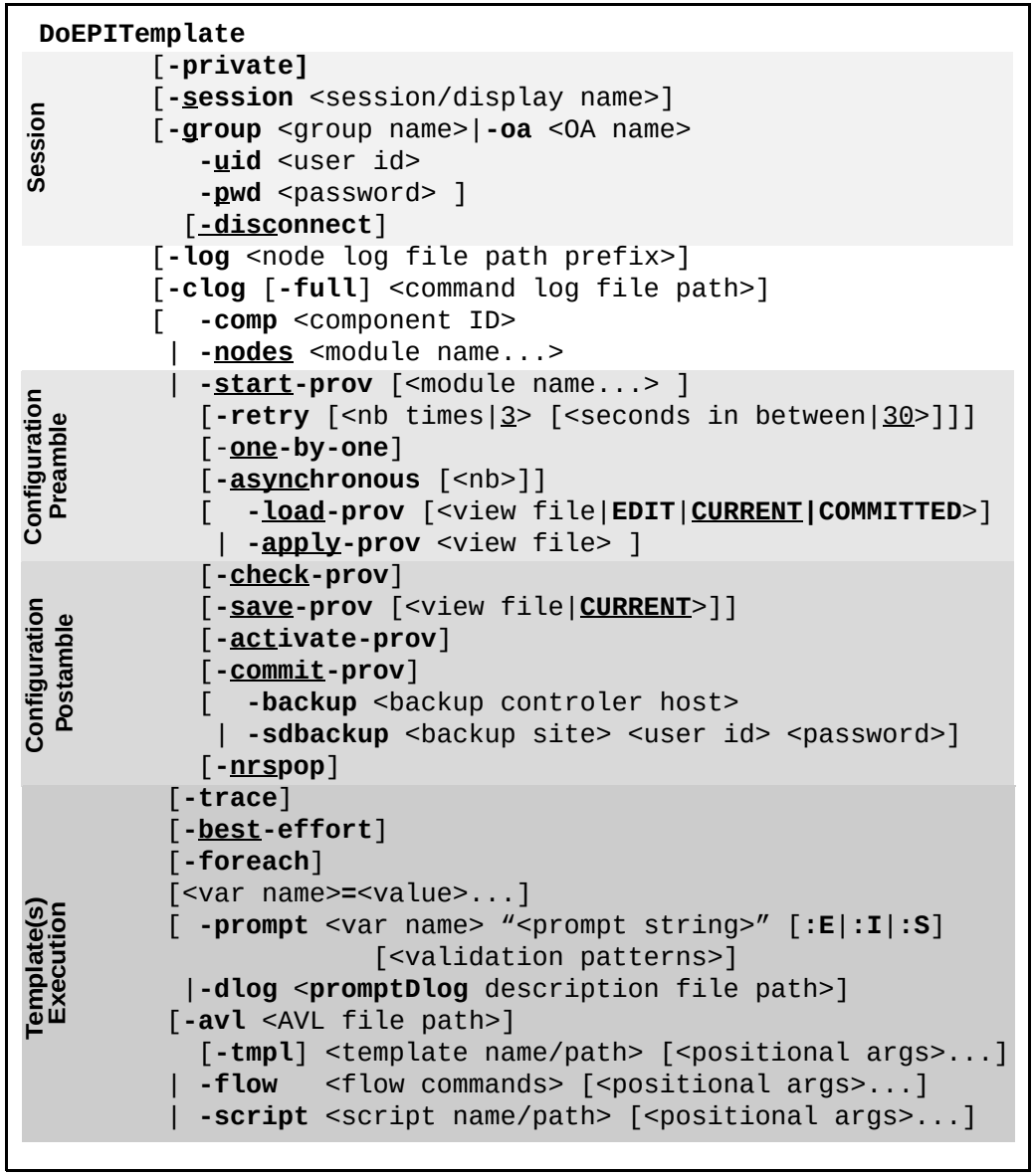


Table 3
DoEPITemplate command options

Option	Description
-private	instructs **DoEPITemplate** to create its own private command session. Since the session is new, you must also specify a group/oa/user-id/password to establish the necessary group connection. If **-private** is not specified, the utility interacts with either the current command session (if called from an MDM user session) or a specific alternate session (as specified by the **-session** option).
-session <display name>	instructs **DoEPITemplate** to use the alternate command session with the specified display name. A group/oa/user-id/password may be required to create the group connection if it is not already established. If you do not use a group/oa/user-id/password is (and without **-private**), the current user/command session is used (as if **-session** "$DISPLAY" had been used). The utility can then interact with your current session and group connections invoked from the Command Console or Component Information Viewer (CIV) Diagnostics.
-group <name> |**-oa** <name> **-uid** <user-id> **-pwd** <password>	if the group/OA connection required by the templates is not guaranteed to be established, use these options to create the connection. You must use these options if you use **-private**. The options identify the group or OA name, user-ID and password. If the options are specified, the script hides these options from its command line. If the connection fails, the utility terminates with an error message: *** Could not establish connection to group/OA <name>. followed by the actual error message from the EPI command.
-disconnect	causes **DoEPITemplate** to disconnect from each Passport node it interacts with when it is finished with each one. Disconnecting from each Passport avoids file descriptor depletion in the process, which may occur when you invoke a template on 1000 or more nodes at a time. Because of the restriction in file descriptor handling, such large node sets can only be processed in "one-at-a-time" or "asynchronous" mode. Bulk mode operations are not able to establish a connection to each node at the same time. See "Multi-node flow (operational mode)" (page 629) and "Multi-node flow (configurational mode)" (page 629).
(Sheet 1 of 11)	

Table 3 (continued)
DoEPITemplate command options

Option	Description
-log <node log file path prefix>	sets the file path and name prefix for the node successful and failed log files. The ".successnodes" and ".failednodes" suffix is appended to complete the name. The default prefix is /tmp/DoEPITemplate.
-clog [**-full**] <command log file path>	requests command level logging. If **-full** is specified, all issued commands (preamble, postamble, house-keeping, and templates) are logged to the specified file. Otherwise, only template/flow related commands are logged. See EPI's NMSCmdOpenCommandLog for more information.
-comp <component name>	specifies a component name that the template may use. The component name is divided into its sub-elements and provided as the following AVL entries to the template (with EM/TOTO LP/1 V35/0 as an example): **mod**the module name (TOTO) **sub**the subcomponent portion (LP/1 V35/0) **comp**<type>one for each subcompnent level **comp**EM -> TOTO **comp**LP -> 1 **comp**V35 -> 0 **comptype**the concatenated component types (EM-LP-V35) You can specify a Passport (type EM) module name with wildcards (*?). The template is then applied to all matching nodes in turn (with **-foreach**) and/or a provisioning session to be managed for each one (with **-start**). Specifying a **-nodes** or **-start** option with the same wild-carded name as the argument has the same effect.
(Sheet 2 of 11)	

Table 3 (continued)
DoEPITemplate command options

Option	Description
-nodes <node names>...	specifies a list of modules to invoke the template (with **-foreach**). You can specify the module names with wildcards (*?). If you use wildcards, all matching Passport nodes are included. You can also specify a module name as @<group name>. All nodes for the specified group are then included. If a node in the list starts with a dot (.) or slash (/), it is taken to be the path of a file containing the names of the nodes to consider (one per line). The first token of each line is extracted to let you use the failednodes file output by **DoEPITemplate** to retry an operation later.
-start-prov <Passport node names>	specifies a list of Passport modules to manage a provisioning session. You can specify the names with wild-cards similarly to the **-nodes** option. If **-foreach** is used, the template will also be invoked for each node taken. For each node, a start prov command will be attempted. If an attempt fails and **-retry** is not specified, the provisioning session will be dropped from all nodes where it was successfully established and the utility will terminate with a failure status and the following error message: *** Failed to start/load/copy configuration on node <name>: <EPI error message> If you specify **-retry**, the utility processes the templates on nodes where the session was successfully opened. The following message is displayed for each module where the session failed: *** Failed to start/load/copy configuration on node <name>1: <EPI error message> *** Ignoring this node. Also, with **-retry**, once all successful nodes are processed, the failed ones are retried up to three times at 30-second intervals.

(Sheet 3 of 11)

Table 3 (continued)
DoEPITemplate command options

Option	Description
-retry [<nb times|3> [<seconds in between|30>]]	(requires **-start-prov**) indicates that any node that fails the start prov must be reattempted (up to three times, with a 30-second delay by default, or as specified), at the end of the normal processing. If the node does not accept after three retries, the node is ignored. The templates are not applied to the node. The node is also ignored if you specify <nb times> as zero.
-asynchronous [<nb, 10>]	(requires **-start-prov**) indicates to execute the configuration preamble and postamble in parallel. See “Process flows” (page 623). The optional count is the number of nodes to be processed in parallel (defaults to 10).
-one-by-one	(requires **-start-prov**) indicates that the entire configuration process should be performed one node at a time. See “Process flows” (page 623).
(Sheet 4 of 11)	

Table 3 (continued)
DoEPITemplate command options

Option	Description

-load-prov [<view name |**CURRENT** __|**EDIT**|**COMMITTED**>]	(requires **-start-prov**)
	For each node, where a provisioning session was successfully started, loads the specified view into the edit view. The view name can be the following special values:
	• **CURRENT**, which uses "copy prov" to map the current view to the edit view
	• **EDIT**, which takes the edit view as it is without loading, applying or copying more information
	• **COMMITTED**, which loads the last committed configuration view. If the loading fails, and **-best-effort** is not specified, the current view is reloaded on all nodes. When the load has succeeded, the provisioning session is dropped from all nodes where it was successfully established, and the utility terminates with a failure status and the following error message:
	`*** Failed to start/load/copy configuration on node <name>:`
	`<EPI error message>.`
	If **-best-effort** is specified, the utility processes the templates on all nodes where the view was successfully loaded. The following message for each module where the session failed:
	`*** Failed to start/load/copy configuration on node <name>:`
	`<EPI error message>`
	`***  Ignoring this node.`

-apply-prov <view name>	same as **-load-prov**, but loads the specified (delta) view with "apply prov" instead of an overwriting load (cannot specify the view name as **CURRENT** nor **EDIT**).
(Sheet 5 of 11)	

Table 3 (continued)
DoEPITemplate command options

Option	Description

-check-prov	(requires **-start-prov**)
	
	For each node with a provisioning session, this command performs a "`check prov`" when all the templates have been executed. If any node fails the check and you do not use **-best-effort**, the current view is re-instated in the edit view for all nodes impacted. The provisioning mode is exited, and **DoEPITemplate** terminates with the following error message:
	`*** Failed to check/save configuration on node <name>:`
	`<EPI error message>`
	
	If you specify **-best-effort**, the failure is ignored and the processing continues. This check is implicit if you specify **-activate-prov** and/or **-save-prov** is specified.

-save-prov [<view name| **CURRENT**>]	(requires **-start-prov**)
	
	For each node with a provisioning session, this command performs a "`save prov`" to the named view. If you specify **CURRENT** this command reuses the view name already associated with the edit view.
	
	If any node fails the save and you do not use **-best-effort**, the current view is re-instated in the edit view for all nodes impacted. The provisioning mode is exited, and **DoEPITemplate** terminates with the following error message:
	`*** Failed to check/save configuration on node <name>:`
	`<EPI error message>`
	
	If you specify **-best-effort**, the failure is ignored and the processing continues. This step is implicit on Passport if you specify **-commit-prov**.

(Sheet 6 of 11)	

Table 3 (continued)
DoEPITemplate command options

Option	Description
-backup <backup controler host>	(requires **-start-prov**) When all provisioning activities are finished and the provisioning session is closed, this command automatically triggers the off-switch backup of the configuration for all affected nodes. If you use **-backup**, the Passport/SNMP Backup tool is used against the specified backup controler host. Using the Passport/SNMP Backup tool performs a view specific backup if a view name cannot be identified. Otherwise, an incremental backup is performed.
-nrspop	(requires **-save-prov**) When all provisioning activities are finished and the provisioning session is closed, this command automatically triggers the population of the NRS database for all affected nodes.
-activate-prov	(requires **-start-prov**) activates and confirms the configuration on all nodes with an established provisioning session. If any node fails the activate/confirm and you do not use **-best-effort**, the current view is re-instated (if not activated) in the edit view for all nodes impacted already. The provisioning mode is then exited and **DoEPITemplate** terminates with the following error message: *** Failed to activate/confirm configuration on node <name>: <EPI error message> If you specify **-best-effort**, the failure is ignored and the processing continues. This check is implicit if you specify **-commit-prov**. ***Note:*** **DoEPITemplate** is not suitable for configuration activations that result in the reloading of the Control Processor (for example, software upgrades). The confirm is not performed correctly when the connection to the Passport node is lost during at activation time.

(Sheet 7 of 11)

Table 3 (continued)
DoEPITemplate command options

Option	Description

-commit-prov	(requires **-start-prov**, implies **-activate-prov**)
	
	When the view is activated, confirmed and saved, this command commits the resulting view.
	
	If a node fails, and you do not use the commit and **-best-effort**, the provisioning mode is exited on all nodes impacted and **DoEPITemplate** terminates with the following error message:
	`*** Failed to commit configuration on node <name>:`
	`<EPI error message>`
	
	If you use **-best-effort**, the failure is ignored and the processing continues.

-trace	if specified, this command passes the **-trace** option to the template execution. The executed command output is provided on output by default. This command also traces the processing of the utility itself.
-best-effort	if specified, passes the **-best-effort** to the template execution (w ignore failure). Also used in postamble processing like **-save-prov** to determine if the processing should continue despite errors. For preamble steps, **-retry** is used to the same effect.
-foreach	invokes all specified templates for each node specified with **-comp** (with wild-carded node names), **-nodes**, or **-save-prov**. Before each template is executed, the **mod** AVL entry is set to the target node.

`<variable name>=<value> ...`	sets the specified AVL variable to the given value. The AVL is passed to each executed template. You can specify multiple settings so that AVL assignments are executed in the specified order. For example, the sequence:
	`mod=node1 dna=$dna2 dlci=$dlci2 \`
	`-tmpl fruni-dlci.tmpl`
	`mod=node2 dna=$dna1 dlci=$dlci1 \`
	`-tmpl fruni-dlci.tmpl`
	
	creates a Frame Relay connection using the same DLCI template invoked on symmetrical AVL specifications.

(Sheet 8 of 11)

Table 3 (continued)
DoEPITemplate command options

Option	Description

-prompt <var name> “<prompt string>” [**:E**|**:I**|**:S**] [<validation patterns>]	prompts you in stand input for the value of the named variable. The specified string is printed as a prompt. **DoEPITemplate** reads the reply and, if a type and validation patterns are specified, **DoEPITemplate** validates it. If the type and validation patterns are invalid, the **DoEPITemplate** prompts again. The expected type of the value can be specified as follows:
	• **:E** for enumerations
	• **:I** for integers
	• **:S** (the default if not specified) for strings
	For enumerations, the patterns to match take the form of a list of legal values separated by blanks or comas. For example,
	`... -prompt lmi “LMI Procedure?” :E \`
	`  “ansi ccitt none” ...`
	For integers, the patterns take the form of a list of individual numeric values or ranges. For example,
	`... -prompt time “Timslots?” :I \`
	`  “1-23,25,27-31” ...`
	Otherwise, the value is a string and you can specify the pattern as a KornShell pattern. For example,
	`... -prompt can “Can interface?” :S \`
	`  “Lp/*\ E1/*\ Chan/*” ...`
	or `... -prompt chan “Chan interface?” \`
	`  “Lp/*\ E1/*\ Chan/*” ...`
	The prompting occurs once at the beginning of the execution. Several special variables can be prompted to control the operations of **DoEPITemplate** as though the corresponding options were specified to its command line. See the table “Special variable names for -prompt, -avl, and -dlog options” (page 621) for a listing of special variables.
	Refer to the **@ASK** and **@CASK** EPI Command template language constructs to prompt users from within the EPI templates, as they are executed.
	DoEPITemplate passes the prompted variable and their entered values to the invoked templates and scripts as if though they were defined with the **=** option.

(Sheet 9 of 11)	

Table 3 (continued)
DoEPITemplate command options

Option	Description
-dlog <**promptDlog** description file path>	indicates that the **promptDlog** utility should be invoked to offer a prompting dialog graphical user interface to the user as described in the specified file. For details about the promptDlog utility, see "The promptDlog Utility" (page 639). The current input AVL of the utility (from the **-avl** option or direct AVL assingments) are provided as the input AVL of the **promptDlog** utilty. The special AVL values indentified in "Special variable names for -prompt, -avl, and -dlog options" (page 621) are also provided to **promptDlog**. If the dialog is positively acknowledged, its output AVL will be loaded into **DoEPITemplate**, becoming its input AVL. If the dialog is cancelled, **DoEPITemplate** immediately terminates without performing any action. This option is the GUI equivalent of the **-prompt** option and both options are exclusive.

-avl <AVL path name>	specifies the full path name of a file containing additional attribute-value pair specifications for the template execution. The file contains one attribute/variable specification per line:
	`# comment lines are ignored`
	`<variable name> <variable value>`
	`...`
	DoEPITemplate loads the file at the beginning of the processing and passes the resulting attributes to the invoked templates and scripts. The variable settings with the **DoEPITemplate** option override the settings from the file. See the table "Special variable names for -prompt, -avl, and -dlog options" (page 621) for a listing of special variables.

(Sheet 10 of 11)	

Table 3 (continued)
DoEPITemplate command options

Option	Description
[**-tmpl**] <template name/path> <positional arguments>	specifies a template to execute with the specified AVL and parameters. You can specify as a full or relative path or a name. If you do not specify a full path, the template is searched in order of the following locations: $HOME/MagellaNMS/tmpl $HOME/MagellaNMS /opt/MagellanNMS/cfg/macros/user/tmpl /opt/MagellanNMS/cfg/macros/user /opt/MagellanNMS/cfg/macros/nms/tmpl /opt/MagellanNMS/cfg/macros/nms /opt/MagellanNMS/lib/epitmpl . If you specify multiple templates, you must specify a multi-valued option, such as **-nodes**, **-tmpl**. Tokens that follow the template specifications are taken to be positional arguments and automatically added to the AVL as entries numbered from one. The first token can be accessed in the template as $1, the second as $2, and so on.
-flow <command flow> <positional arguments>	Instead of specifying one or more template files to execute, use **-flow** to indicate an inline command flow (string). Unlike **-tmpl**, only one flow can be specified and this cannot be used in conjunction with **-tmpl**. The positional arguments that follow are processed the same way as for **-tmpl**.
-script <script name/path> <positional arguments>	Instead of executing an EPI command flow or template, this command executes the specified script as a separate process. The varBinds are passed on to the script as a set of environment variables of the form: **param**_<varBind name> For example, the module name varBind (**comp**EM) constructed by the **-comp** option results in the environment variable: **param_comp**EM -> <module name>. The specified positional arguments are also passed on as command line arguments to the script invocation.
(Sheet 11 of 11)	

The table “Special variable names for -prompt, -avl, and -dlog options” (page 621) describes the special variable names.

Table 4
Special variable names for -prompt, -avl, and -dlog options

Variable name	Use (equivalent/overriden command line option)
@start or **@nodes**	same as **-start** or **-nodes** to determine the node(s) to apply the templates to
@load	same as **-load** to specify the Passport view name to load
@apply	same as **-apply** to specify the Passport view file to apply to the current configuration
@save	same as **-save** to specify the resulting Passport view name for the configuration
@group or @oa	same as **-oa** or **-group** to determine the group or OA to connect to
@uid	same as **-uid** to specify the user ID to authenticate with
@pwd	same as **-pwd** to specify the matching password
@comp	same as **-comp** to specify the component ID to apply the template to
@avl	same as **-avl** to specify an AVL file with additional attribute specifications
others	other variables are passed with their value to the called template/ scripts as substitution variables (AVL).

Interaction with command session

DoEPITemplate requires an active command session to establish device connections, and forward commands to them. By default, as when it is invoked from the Command Console or within a Preside Multiservice Data Manager (MDM) user session, **DoEPITemplate** interacts with the current session that is mapped to the current DISPLAY environment variable).

DoEPITemplate uses the existing group and OA connections. For example, **DoEPITemplate** can use connections established through the Command Console or through the macro invocation of the Connect Console.

See 241-6001-301 *Preside MDM Customization Administrator Guide*. **DoEPITemplate** can get the utility to create the necessary connection with its **-group**/**-oa**, **-uid**, and **-pwd** options.

When **DoEPITemplate** is invoked outside of a normal session, such as from CRON, a private command session must be initialized. The private command session can be initialized as follows:

- in the usual manner for command macros by calling the **DoEPITemplate** utility (or a script that invokes it) as a command line argument to the `cmcwrap` (`/opt/MagellanNMS/bin/cmcwrap`) utility.

  ```
  /opt/MagellanNMS/bin/cmcwrap \
     /opt/MagellanNMS/bin/DoEPITemplate ... \
                 -group <group> -uid <user id> \
                 -pwd <password> ...
  ```

  The necessary group/OA connections must then be established either before **DoEPITemplate** is invoked (as from a script using the `cmccmd` utility) or with the utility itself through its **-group**/**-oa**, **-uid** and **-pwd** options.

- simpler than `cmcwrap` if only **DoEPITemplate** is used, the utility can create its own private command session by specifying the **-private** option.

  ```
  /opt/MagellanNMS/bin/DoEPITemplate ... \
                 -private \
                 -group <group> -uid <user id> \
                 -pwd <password> ...
  ```

  DoEPITemplate initiates the session and automatically terminates it. You must use the **-group**/**-oa**, **-uid**, and **-pwd** options to establish the necessary connection.

- by initiating a private command session on the side, for example, with a separate EPI script calling the `NMSCmdSession` command. You can also specify the corresponding command session name (typically the matching `DISPLAY` environment variable value) with **`DoEPITemplate`** -session option.

```
# assuming a DtKsh EPI script, start
# a private command session
NMSCmdSession -display PrivateSession$$ start

/opt/MagellanNMS/bin/DoEPITemplate ... \
              -session PrivateSession$$ \
              -group <group> -uid <user id> \
              -pwd <password> ...
...
```

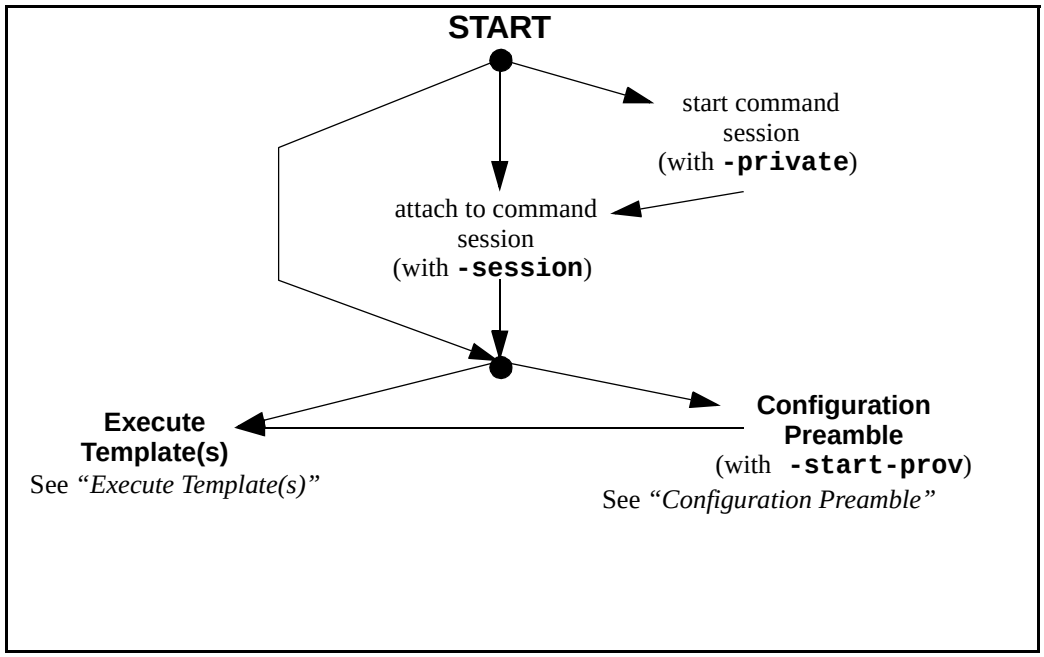
  This can be done to get multiple**`DoEPITemplate`** invocations to share the same command session servers and device connections. It is important that the various invocations do not try to configure the same Passport nodes as the commands performed by both invocations will inter-mingle and cause an invalid device configuration.

Process flows

`DoEPITemplate` supports several different processing flows. It supports the execution of the following:

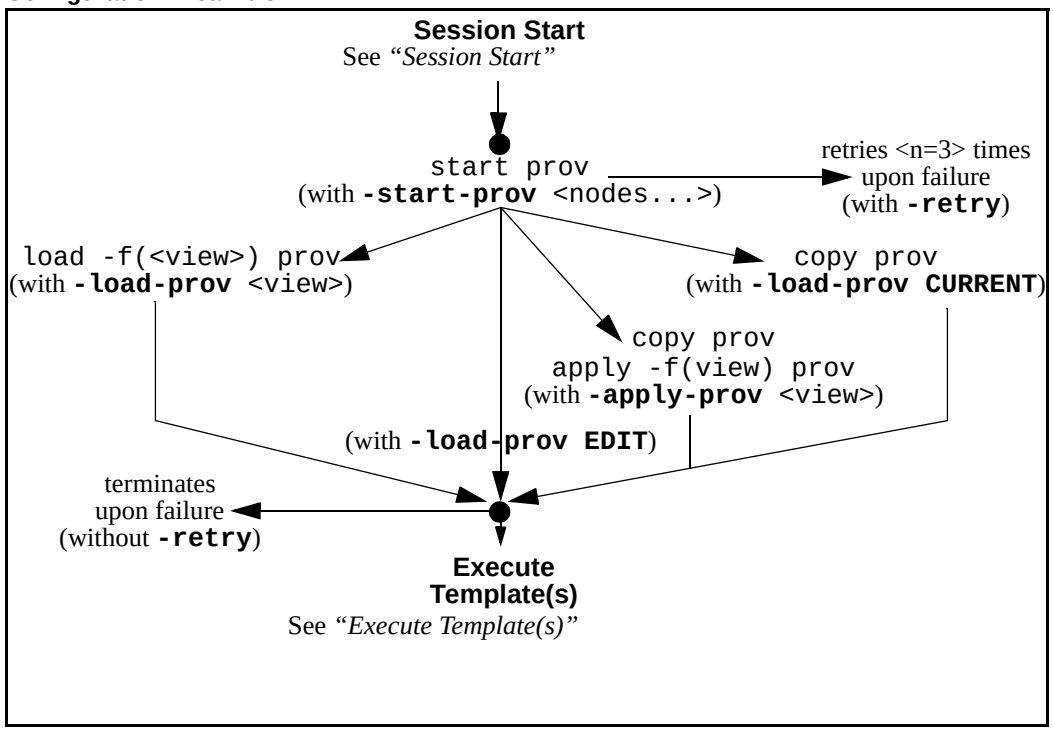
- operational command templates (no provisioning commands and no **`-start-prov`**) on one of more DPN or Passport nodes
- configurational common templates on one or more Passport nodes, where **`DoEPITemplate`** handles the provisioning session control. Provisioning Session control is made out of the following phases:
  - — a session start, common to both operational and configurational mode, where a new private command session is established, if required. See “Interaction with command session” (page 621), and the figure “Session Start” (page 624).

Figure 11
Session Start



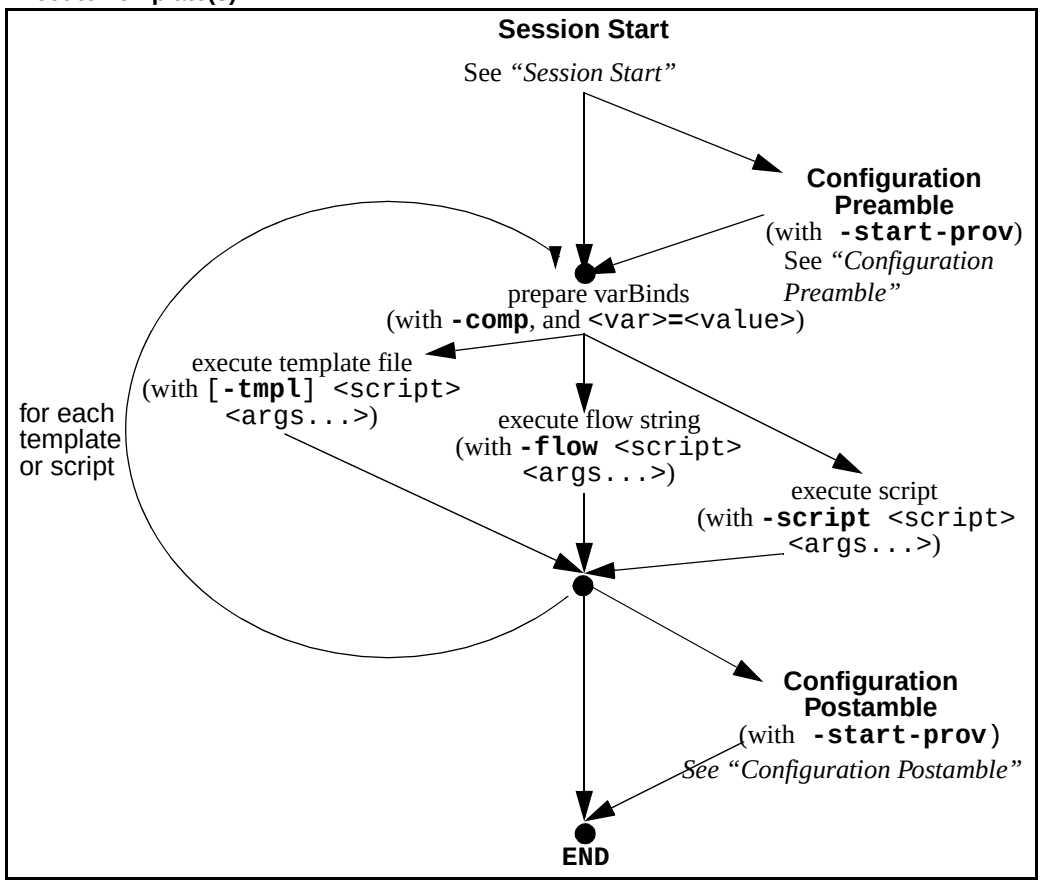
— a preamble consisting of the **`start`**, **`copy`**, **`load`**, and or **`apply prov`** commands, as shown in the figure "Configuration Preamble" (page 625).

Figure 12
Configuration Preamble



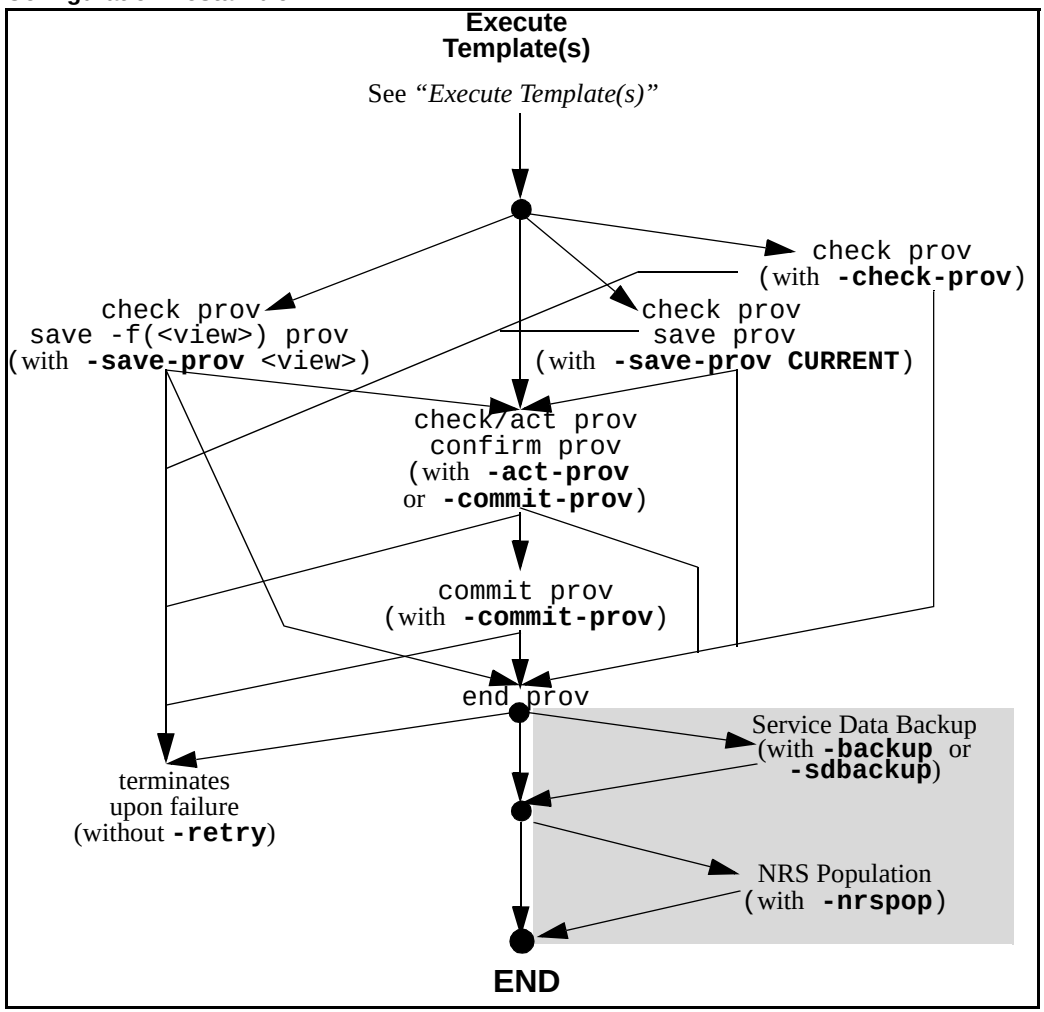
- the execution of the specified templates, as shown in the figure "Execute Template(s)" (page 626).

Figure 13
Execute Template(s)



— a postamble consisting of the **`check`**, **`save`**, **`activate`**, **`confirm`**, **`commit`** and **`end prov`** commands (with the triggering of the NRS population as a special extension)

Figure 14
Configuration Postamble

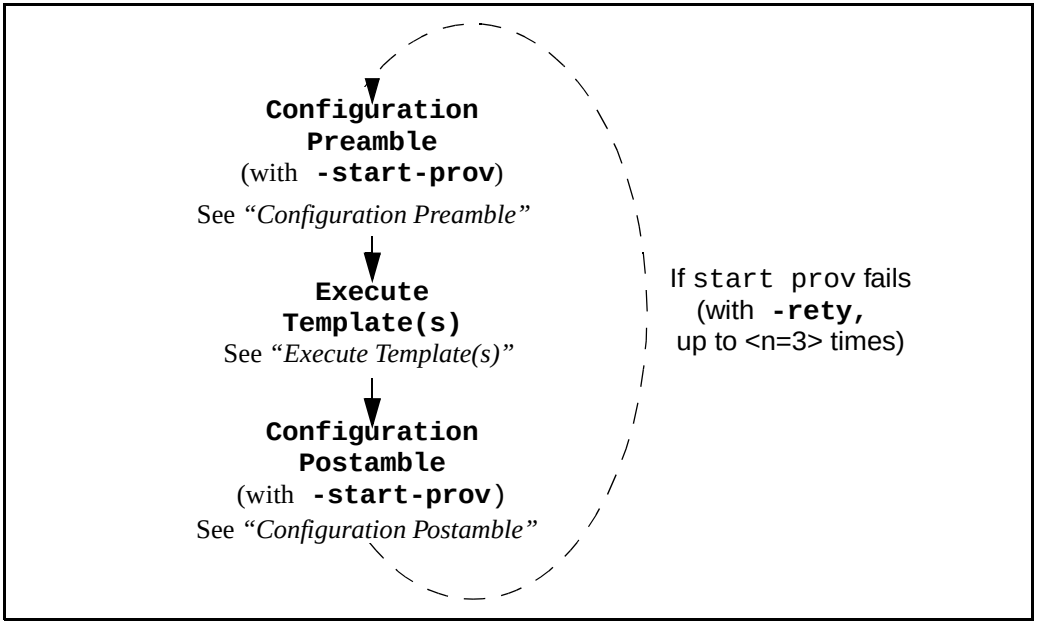


Single node flow

The process flow applying to a single node (**-nodes**, **-comp**, or **-start-prov** arguments expanding to a single node) is obvious in both cases. In operational mode (no **-start-prov**), the specified templates are run in

their specified order, with varbind (re)assignments in-between. In configurational mode, the preamble is executed as requested, followed by the templates, and the postamble, as requested.

Figure 15
Single Node Configuration Flow



For example;

```
DoEPITemplate  -private \
               -start NODER16 -load CURRENT  \
               -activate \
               lp=3 ds1=0 chan=0 time1=0 time2=11 \
               -tmpl AddDs1Chan.tmpl
```

In a private command session, Adds DS1 Channel LP/3 DS1/0 CHAN/0 with 12 timeslots to the current configuration of node NODER16 and activates it.

Multi-node flow (operational mode)

Applied to multiple nodes, **DoEPITemplate** supports several different flows. For operational templates, you can execute different templates on different nodes by changing the appropriate varBind:

```
DoEPITemplate  mod=NODER16 sub="lp/3 ds1/0" \
               -tmpl DS1LoopBackTest.tmpl \
               mod=NODER10 sub="lp/10 V35/2" \
               -tmpl XLoopBackTest.tmpl
```

Performs a DS1 loopback test on EM/NODER16 LP/3 DS1/0 and a V35 loopback test on EM/NODER10 LP/10 V35/2.

The same template can be automatically applied, in sequence, to several nodes (**-nodes** or **-comp** mapping to multiple devices) automatically with the **-foreach** option. **DoEPITemplate** automatically defines the "**mod**" varBind value to the current module:

For example:

```
DoEPITemplate  -nodes @EAST_GROUP
               -foreach \
               -best-effort \
               -flow "$mod tidy prov"
```

Invokes the tidy prov command on all nodes in the EAST_GROUP passport group. The template/flow is executed in best-effort mode, ignoring failures if they occur.

In this example, the individual nodes names may be directly listed (**-nodes** NODE1 NODE2 NODE3 ...) or wild-carded (**-nodes "**NODE*"). The name is quoted to prevent the shell from expanding the * into the matching file names). Similarly, **-comp** may be used to identify multiple nodes (for example, **-comp "**EM/NODE* PROV").

<- command
session
<- preamble
<- postamble
<- template

Multi-node flow (configurational mode)

In configurational mode, there are three processing flows over multiple node selections (**-start-prov** arguments mapping to multiple nodes). The processing flows are as follows:

- bulk
- one-node-at-a-time

- asynchronous processing

Bulk processing

Bulk processing is the default processing flow over multiple node selections. Bulk processing functions as follows:

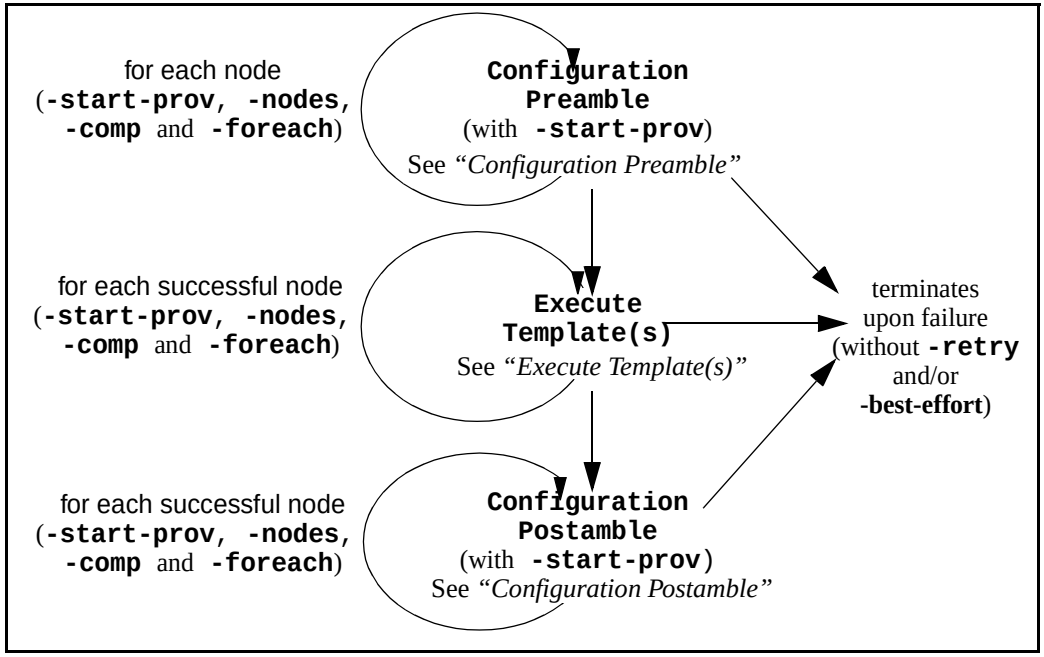
- The preamble is performed as specified on every node in sequence. The `start prov` and the specified view is loaded, copied, and applied on each node.
- The templates are executed in sequence. If you specify **`-foreach`**, the templates are invoked on each specified node. The name of the current module is passed to the template as the "**`mod`**" varBind value.
- The postamble is executed as specified on every node in the following sequence:
  - A `check prov` is performed on every node.
  - The configuration is saved on all nodes, if requested.
  - The configuration is activated and confirmed on all nodes, if requested.
  - The configuration is committed on all nodes, if requested
  - The `end prov` is performed on each node.

If you do not specify **`-retry`**, the process is stopped upon failure of any step and the unactivated configuration sessions are undone and ended. With the **`-retry`** command, the processing continues and failed nodes (`start prov` only) are re-attempted when the first pass is complete. If the number of reattempts specified with an option of zero, the failed node is ignored.

If you do not specify **`-best-effort`**, all configured nodes must pass the `check prov`. If one node fails, the provisioning session is abandoned on all devices. No changes are saved or activated.

The figure "Bulk Configuration Flow" (page 631) describes bulk configuration.

Figure 16
Bulk Configuration Flow



For example:

```
DoEPITemplate  -start NODER16 NODER10 \
               -load CURRENT \
               -save 01052100 -activate \
               -nrspop \
               mod=NODER16 fruni=120 dlci=23 \
                           dna=12345123456 \
               -tmpl CreateGoldDlci.tmp \
               mod=NODER10 fruni=1200 dlci=144 \
                           dna=12345654321 \
               -tmpl CreateGoldDlci.tmp
```

Ensures a provisioning session on NODER16 and NODER10 and creates a Frame Relay connection between both nodes. Then the resulting configuration on both nodes are checked.

The Bulk Configuration process flow is suitable for "correlated" configuration tasks, such as the Frame Relay connection example. The process flows guarantees that the `start prov` is possible on the applicable nodes before the configuration is performed, and the configuration is checked on all nodes before it is saved or activated.

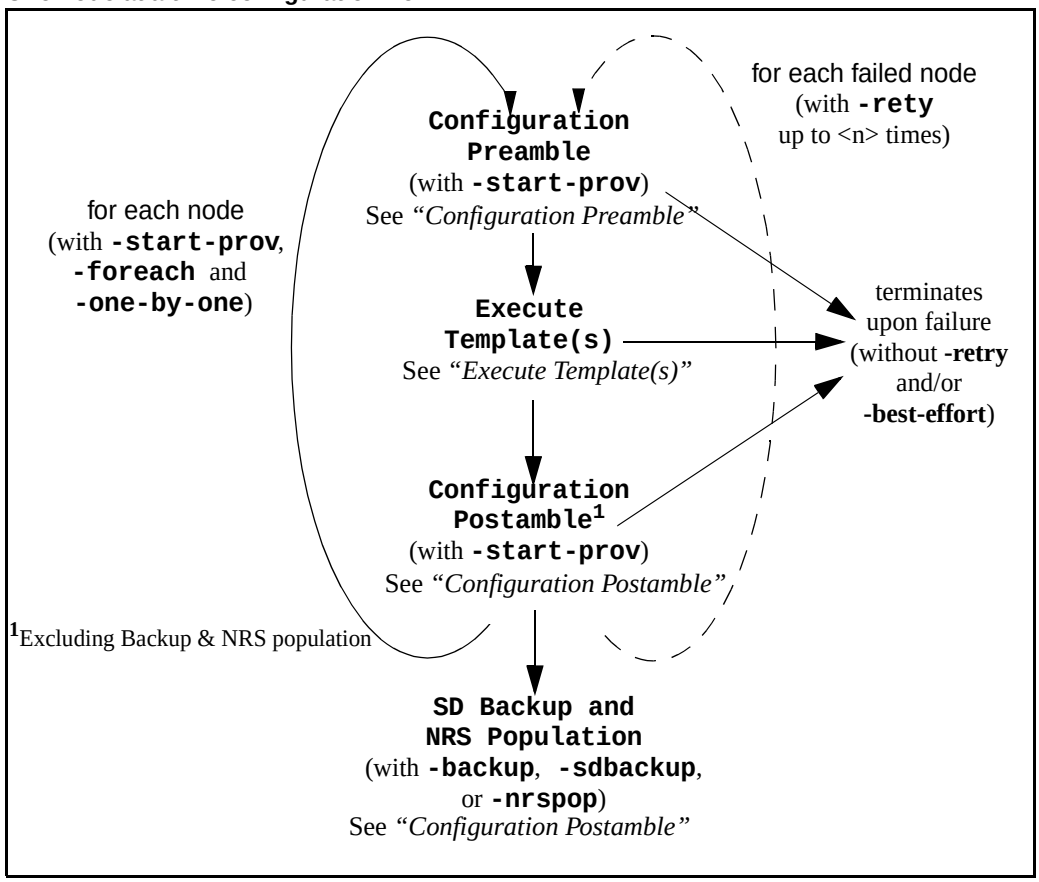
One node at a time

If you specify **`-one-by-one`** and **`-foreach`**, the flow is performed fully for each node, one at a time:

- For each specified node (**`-start-prov`**)
  - Perform preamble as specified (`start`, `load`/`copy`/`apply`).
  - Execute the templates. The current node is provided to the template as the "**`mod`**" varBind value.
  - Perform the postamble as specified (`check`, `save`, `activate`/`confirm`, `commit`, `end`).
- When all nodes are completed, perform the Service Data Backup and NRS population on the successful nodes, if requested.

If any step fails and you do not specify **`-best-effort`**, the unactivated configuration changes and session of the current node are undone. If you specify **`-retry`**, all nodes that failed the `start prov` are re-attempted when the first pass at all nodes is completed. If the number of reattempts specified has the option set to zero, the failed nodes are ignored. There is no guarantee that the `start prov` will succeed on all nodes before the templates were executed and the configuration saved and activated. There is also no guarantee that the configuration was checked successfully on all nodes before it is activated.

Figure 17
One node at a time configuration flow



For example:

```
DoEPITemplate  -start @GROUPALL -retry \
               -foreach -one-by-one \
               -load CURRENT \
               -save 01052100 -activate -commit \
               -tmpl AddUser.tmp OPER2 "OPER2pwd" \
```

`network systemAdministration`

For each node in group GROUPALL, one by one, starting the provisioning session against the active configuration, executes the AddUser template with the specified arguments passed as the $1, $2, $3, and $4 varBind values. The resulting configuration is checked, saved, activated and committed. Any node that may have failed to start prov is reattempted at the end.

This processing flow applies the same configuration to multiple nodes in sequence. Since the provisioning session is taken one node at a time, the locking is reduced.

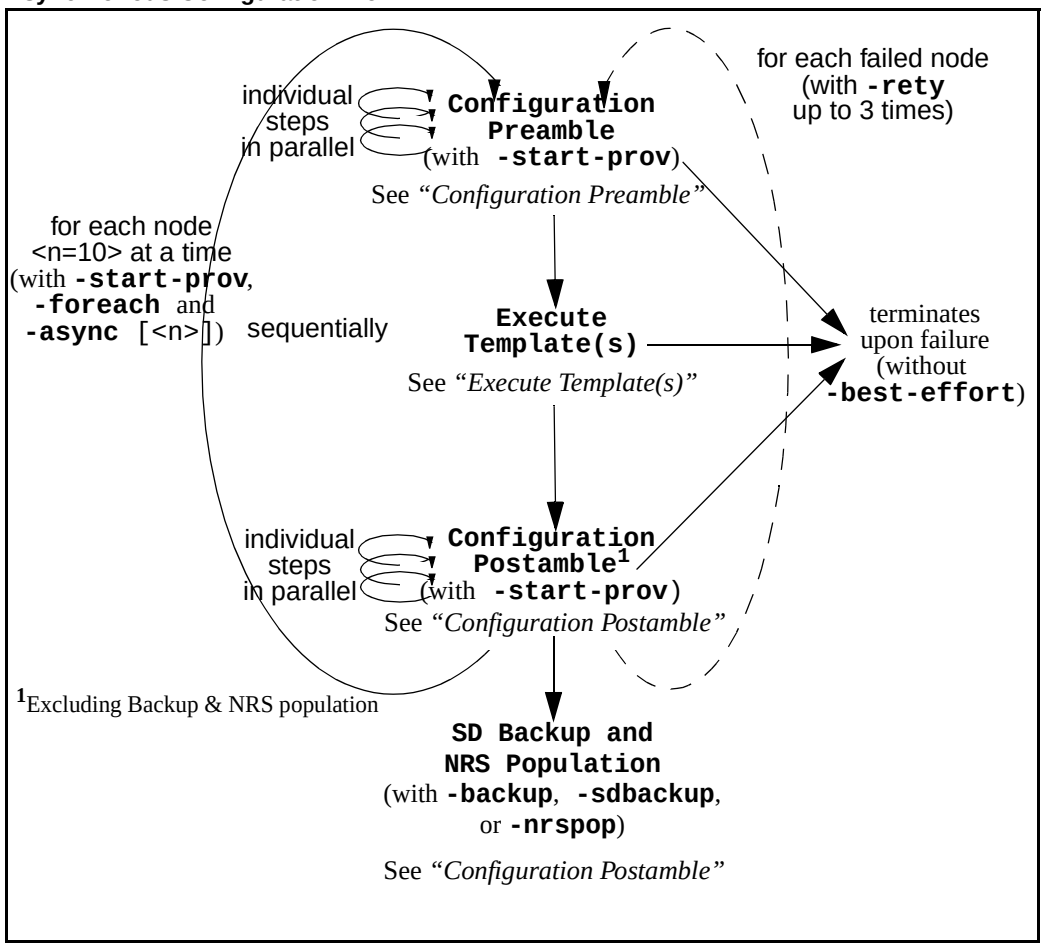
Asynchronous

The asynchronous process flow is enabled with the **`-asynchronous`** option. The processing of the preamble and the postambles are performed in parallel:

- For each specified node, `<n>` node at a time, where `<n>` is the optional argument to **`-asynchronous`**, and defaults to `10`
  - Perform the preamble, as specified, in parallel. `start prov` for all, then load/`copy`/`apply` for all.
  - Execute the specified templates in sequence, not in parallel.
  - Perform the postamble, as specified, in parallel. Perform `check` for all, `save` for all, `activate`/`confirm` for all, `commit` for all, and `end prov` for all.
- When all nodes are completed, the process flow launches the Service Data Backup and NRS population on the successful nodes, if requested.

Reaction to failures of a step is controlled by the **`-retry`** and **`-best -effort`** options. If you use **`-retry`**, nodes that failed to `start prov` are reattempted in parallel at the end. If the number of reattempts specified with the option is set to zero, the failed nodes are just ignored. If you specify **`-foreach`**, the templates are executed for each node currently in provisioning node in that processing chunk (`<n>`).

Figure 18
Asynchronous Configuration Flow



For example:

```
DoEPITemplate -start @GROUPALL -retry \
              -async 20 -foreach \
              -load CURRENT \
              -save 01052100 -activate \
              -tmpl AddUser.tmp OPER2 "OPER2pwd" \
```

```
                                network systemAdministration
```

Processing is performed, in parallel, for every set of 20 nodes.

Since the preamble and postamble are performed in parallel, the on-switch provisioning mode locking is minimized. There is no guarantee that all nodes have accepted the start-prov before the templates are executed. There is also no guarantee that the configuration has checked on all node

Additional examples

This section contains the following additional examples:

- “Prompting” (page 636)
- “Specifying a AVL file” (page 637)

Prompting

This example shows how to invoke **DoEPITemplate** in prompting mode to configure a Frame Relay interface:

```
DoEPITemplate  -retry \
               -load CURRENT \
               -save CURRENT -activate \
               -prompt @start “Passport node?”
                      “?*” \
               -prompt fruni “FrUni number?” :I \
                      0-65535 \
               -prompt lmi “Lmi procedure?” :E \
                      “none ansi itu” \
               -prompt interface “DS1 Channel?” \
                      “Lp/*\ Ds1/*\ Chan/*” \
               -tmpl AddFrUni.tmp
```

Prompts you for the the node to configure. The special variable @start informs **DoEPITemplate** to start a provisioning session on the named nodes. You are also prompted for the following:

— node name, which must have at least one character (?*” Ksh pattern)

— a FrUni instance number in the 0-65535 range

— an LMI procedure from a fixed set of values)

— a Passport DS1 channel component name

The AddFrUni.tmp template is then invoked, with the replied variable values to create the necessary component structure.

For an example of the use of the -dlog option for graphical user-interface prompting, see “The promptDlog Utility” (page 639).

Specifying a AVL file

This example shows how to use a separate AVL file to provide parameters to the template. It is a modified version of the prompting example. Assuming a text file called `/tmp/fruniparams.data`.

The call with the following contents

```
@start NODER16
fruni 1220
lmi none
interface Lp/12 Ds1/2 Chan/0
```

```
DoEPITemplate -retry \
              -load CURRENT \
              -save CURRENT -activate \
              lmi=none \
              -avl /tmp/fruniparams.data \
              -tmpl AddFrUni.tmp
```

does the same as the prompting example, except that it extracts values from the provided AVL file. The lmi value is given a default value on the command line. The **`-avl`** and **`-prompt`** based variable specifications override those given on the command line.

Appendix B
The promptDlog Utility

This appendix describes the promptDlog utility.

The promptDlog utility lets you create a configurable data entry form GUI to prompt users for a number of parameters. The values for these parameters are then made available for use in EPI templates and other scripts. You can configure the dialog for single or multiple pages. When using multiple pages, the promptDlog utility lets you create wizards with specific transition rules from page to page.

The promptDlog supports a range of input controls from basic (for example, text fields, radio buttons, and check boxes) to advanced (for example, date/time entry and component specification).

The value of each control is passed to the calling utility as an attribute value (AVL) flow, using one attribute and value specification per line. The AVL can also be loaded into another utility and is compatible with the DoEPITemplate AVL input file.[1] You can also use an input AVL, whether by file or explicit variable setting on the command line, to specify initial field values within the form. Input AVL fields not used in the dialog are passed through to the output AVL as hidden variables.

When you confirm a dialog by clicking the OK button, the promptDlog utility returns a code of 0 and outputs the AVL to either standard output or the file identified on the command line. If you cancel the dialog, the utility returns a code of 1 and does not output the AVL.

[1] Specifying the -dlog option with the DoEPITemplate command implicitly starts the promptDlog utility.

Command line options

You access the promptDlog utility from the /opt/MagellanNMS/bin directory. The figure “The prompDlog command” (page 640) shows the promptDlog command and list of available command line options. The table “promptDlog command line options” (page 640) provides details for each command line option.

Figure 19
The prompDlog command

```
promptDlog
       -desc <dialog description file>
       [-out <output AVL file name|STDOUT|NONE>]
       [-avl <initial AVL file name>]
       [-start <start page name>]
       [-silent]
       [-inline]
       [<variable name>=<value> ...]
```

Table 5
promptDlog command line options

Option	Description
-desc <dialog description file>	Mandatory Identifies the dialog description file to use.
-out <output AVL file path|**STDOUT**|**NONE**>	Specifies a file name for the dialog’s output AVL. By default, or if STDOUT is specified, the AVL prints on the standard output stream. If NONE is specified, no AVL is output.
-avl <input AVL file path>	Specifies a file name for an input AVL. By default, there is no input AVL.
-start <start page name>	In multi-page mode, identifies the page name where the dialog will start. By default, the first page in the description file is used as the start page.

(Sheet 1 of 2)

Table 5 (continued)
promptDlog command line options

Option	Description
-silent	When a field in the form is found to be invalid, the utility rings the bell. This option turns off this functionality.
-inline	Specified when sourcing the promptDlog code in another script. The dialog can then be controlled by code. See “Inline execution and action support calls” (page 700) for more information on inline execution.
<variable name>=<value>	Specifies a variable for the input AVL.
(Sheet 2 of 2)	

Dialog description file format

The dialog description file is similar in format to other MDM Xt/Motif resource files (for example, the Toolset Menu files). The file consists of a number of blocks separated by blank lines. Each line of the block specifies a resource using the following format:

```
<resource name>: <value>
```

Example:

```
pDpage:          one

labelString:    User ID:
pDtext:         uid
pDtextcolumns:  8
pDtextPatterns: +(?)
pDhelp: User Name (1 to 8 ASCII characters)

labelString:        Password:
pDpassword:         pwd
pDtextcolumns:      16
pDpasswordPatterns: ?????*(?)
pDhelp: Password (5 to 8 ASCII characters)
```

The description file uses the following conventions:

- The resource name is case sensitive.

- You can continue long lines by adding a terminating back-slash (\) before continuing on the next line.
- Any occurrence of a `\n`, `\t`, or `\b` sequence is substituted by a new line, tab, and blank character, respectively.
- Comments are indicated with an initial # or ! character on a line.

There are types of records in a dialog descriptor file:

- "Page descriptions" (page 642)
- "Field descriptions" (page 642)

Page descriptions

The first record in a page description block is the page description record. This record begins with the "pDpage:" resource name followed by a value for that resource. Subsequent records in the block are field description records. These records provide descriptions for fields on the page. If you omit the page description block, or specify a single page description block, the promptDlog utility works in single page mode. Otherwise, the utility works in multiple page mode.

For details about available field descriptors for this block, see the following topics:

- "Page and Dialog Control" on page 646

Field descriptions

The first record in a field description block is the field description record. This record begins with the "labelString:" resource name followed by a value for that resource.

For details about available field descriptors for this block, see the following topics:

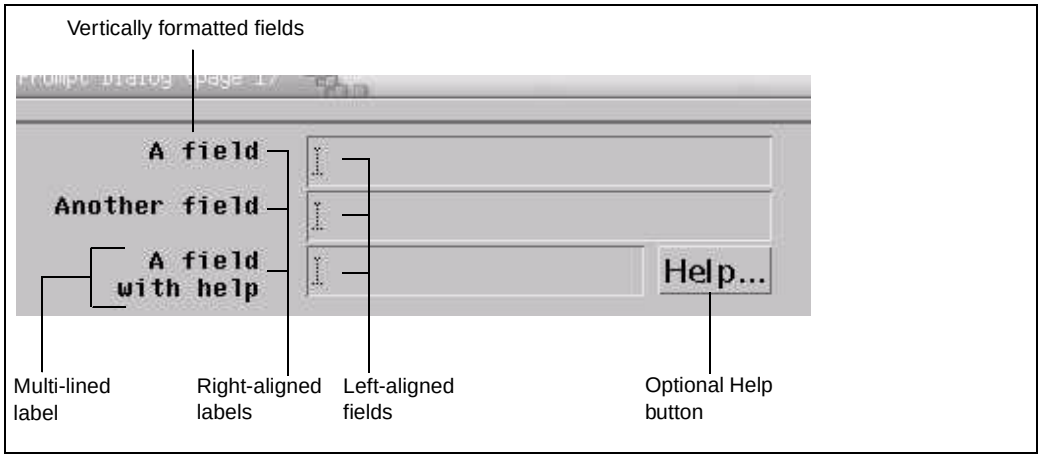
- "Title and separator" on page 658
- "Text entry field" on page 660
- "Scrolled text field" on page 663
- "Password field" on page 666

Field layouts

A dialog field usually consists of a label on the left and a corresponding specialized field on the right. The layout of the fields is vertical–they appear one below the other, in the order that they are specified in the description file.

Labels in the dialog are right-aligned to the longest label; specialized fields are left aligned. Labels can contain multiple lines (with an embedded `\n` in the description file). The label part of title fields are ignored in this alignment. See the figure “Basic field layout” (page 644) for a sample layout.

Figure 20
Basic field layout



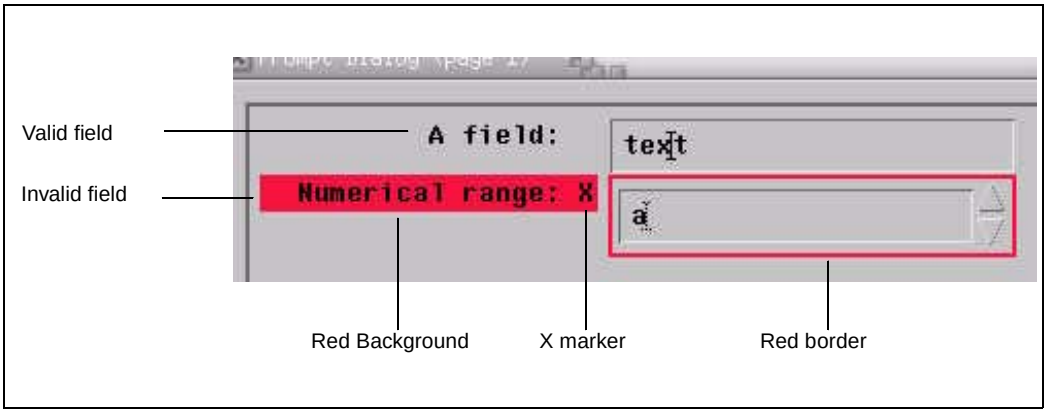
Help button

You can configure a field with specific context help information by using the `pD<field type>Help:` line. When you do, a Help push-button is added to the right of the field. Clicking the Help button displays the associated help information in a separate dialog.

Valid and invalid fields

The promptDlog utility performs a validity check on the value of fields when these field lose their keyboard focus. Checks are done automatically for certain field types (for example, dates and numerical ranges). Checks for more general field types are based on the provided patterns in `pD<field type>Patterns:` lines. When all fields have been positively validated, the OK confirmation button is enabled. Otherwise, invalid fields are highlighted. Invalid fields have an X is added to their label, and their background and borders change to red. See the figure “Field validation feedback” (page 645).

Figure 21
Field validation feedback



Each field in the description file is given an AVL name for the variable that will contain its value in the output AVL. For this reason, each field name in the description file should be unique. If you specify an input AVL when you launch the promptDlog utility, the values for those variables in the input AVL supersede any values specified in the description file.

KSH utility functions

You may require some KSH code in the description of the dialog (for example, page transition rules, enter and leave actions, and cancel and confirm actions). You can source the entire dialog into another DTKSH script rather than have a separate executable. To help with this, promptDlog provides a number of KSH utility functions that you can invoke from the added KSH code. For example, you can use `pDsetValue` to program the value of a field. For more information on these calls, see “Inline execution and action support calls” (page 700).

pDpage descriptors

This section describes the following pDpage descriptors:

- "Page and Dialog Control" (page 646)
- "Title and separator" (page 658)

The table "promptDlog description file directives: page" (page 654) provides a complete list of description resources applicable to a `pDpage:` block.

Page and Dialog Control

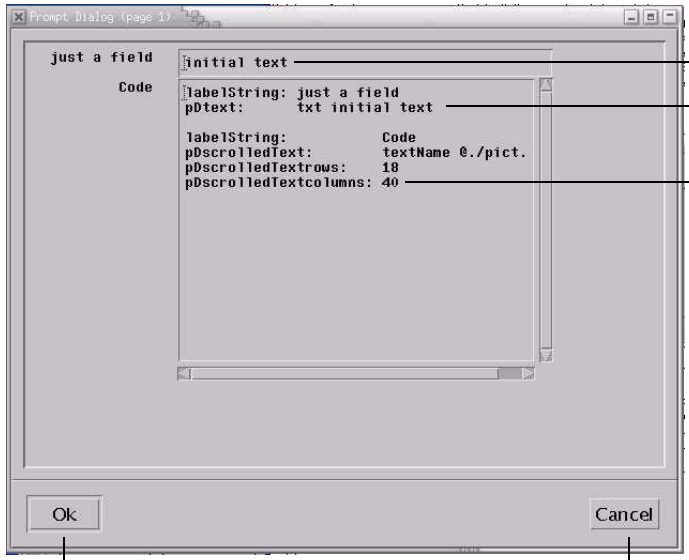
You can use single or multiple page modes for the prompting dialog.

Single page mode

By default, the prompting dialog uses single-page mode. See the figure "Sample single page prompt dialog" (page 647). The single page contains those fields that have been defined in the description file. The fields display vertically in a scrolled window. In single page mode, the dialog has an Ok and Cancel button (the Help button is optional). The Ok button is enabled only when all fields are validated against any provided patterns.

To specify page resources (`pDpageTitle:`, `pDpagePixmap:`, `pDpageOkLabel:`, `pDpageCancelLabel:`, `pDpageHelp:`, `pDpageLeaveAction:`, `pDcode:/pDcodeEnd:`, and `pdPageInclude:`) in single page mode, provide a single `pDpage:` description block at the beginning of the description file.

Figure 22
Sample single page prompt dialog



Description:

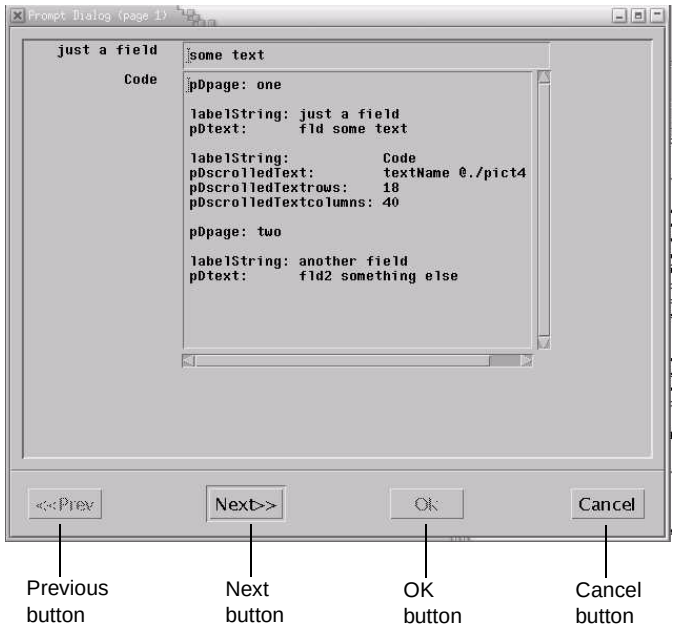
```
labelString: just a field
pDtext:      txt initial text

labelString:            Code
pDscrolledText:         textName @./pict3.txt
pDscrolledTextrows:     18
pDscrolledTextcolumns:  40
```

Multiple page mode

You can also create a multiple page dialog. See the figure “Sample multiple page prompt dialog” (page 648). Multiple pages are chained together in a “wizard” fashion. Each page is defined in the description file by a page description record.

Figure 23
Sample multiple page prompt dialog



Description:

```
pDpage:    one

labelString:    just a field
pDtext:         fld some text

labelString:              Code
pDscrolledText:           textName @./pict4.txt
pDscrolledTextrows:       18
pDscrolledTextcolumns:    40

pDpage:    two

labelString:    another field
pDtext:         fld2 something else
```

In multi-page mode, two additional buttons display in the dialog. The Next button transitions the next page in the flow. The Next button is enabled when all fields in the current page are validated against any available patterns and the page is not the last one in a flow. The Previous button displays the previously visited page.

Page flow

`The pDpage`: record specifies a name for a page. You can specify more than one name for a page. This record also defines the page transitions through a set of next page rules (`pDpageRule:`). Page rules are sections of Ksh code that can query the current value of the AVL (with the `pDitemValues[<var. name>]` associative array) and set the `pDnextPage` variable to the name of the next page or to an empty string. An empty string has the same effect as not setting the variable. The first rule, in the order specified in the description file, sets the variable. If you do not specify any rules, the next page specified in the description file is used. Backtracking through pages using the Previous button always brings back the pages in the reverse order they were visited.

Setting a stop page in the page flow

In multiple mode, the OK button is only enabled for the last page, as specified in the description file. However, you can make any page a “stop” page. To do so, specify a value for the page name in the `pDpageStop:` resource line. The page becomes a stop page if the transition to it uses the value specified in the `pDpageStop:` resource line. This allows pages to be intermediate in some flows but final in others. Similarly, you can make a page the last one in a flow by specifying a page name on the `pDpageLast:` line. Using `pDpageLast:` has the same effect as `pDpageStop:` except the Next button is always disabled. For example, assume the current page has two names specified in its `pDpage:` line; —*accessParm* and *accessParm_end* and that *accecssParm_end* is the current value of a `pDpageLast:` line. If the previous page transitions to the current page (`pDpageRule:` line) by specifying *accessParm* as the value of the `pDnextPage` variable, the current page is considered an intermediate page. If the `pDnextPage` variable was set to *accessParm_end*, then the current page is considered final. The `pDpageStop:` option is useful for creating a dialog with one or more mandatory pages and a number of optional pages (for example, for advanced

parameters). The last mandatory page and the subsequent optional pages would all be given a `pDpageStop:` line with their page names so that you can confirm the dialog at any point.

Entering and leaving a page in the flow

You can define an enter (`pDpageEnterAction:`) and/or leave (`pDpageLeaveAction:`) action for a page. These actions are sections of Ksh code that execute when the page is entered and left. The page leave action also occurs when a page in a dialog is confirmed.

In the code for these actions, you can access the `pdPageTransition` variable. This variable takes on the following values:

- `NEXT` when the call is the result of forwarding through the dialog with the Next button
- `PREV` when backtracking with the Previous button)
- `OK` (Leave action only) when the call is the result of the dialog confirmation

A typical use for these actions would be to reset some field values when a page is entered or backtracked or to set “hidden” variables to track which pages are visited.

Two more actions can be invoked when the dialog is confirmed by pressing the OK button (`pDconfirmAction:`) or cancelled (`pDcancelAction:`) with the Cancel button.

To support all these action-based specifications (`pDconfirmAction:`, `pDcancelAction:`, `pDpageEnterAction:`, `pDpageLeaveAction:`, and `pDpageRule:`), you can source one or more Ksh code definition files into the script at startup by using the `pDinclude:` statement. Alternatively, you can include the Ksh code in the description file at any point between the `pDcode:` and `pDcodeEnd:` lines.

Titles

By default, no titles or images display in the dialog. However, you can specify a title for each page (`pDpageTitle:`). The title appears at the top of the dialog and in the window frame’s title bar. See figure “Sample prompt dialog with page title and page help” (page 652).

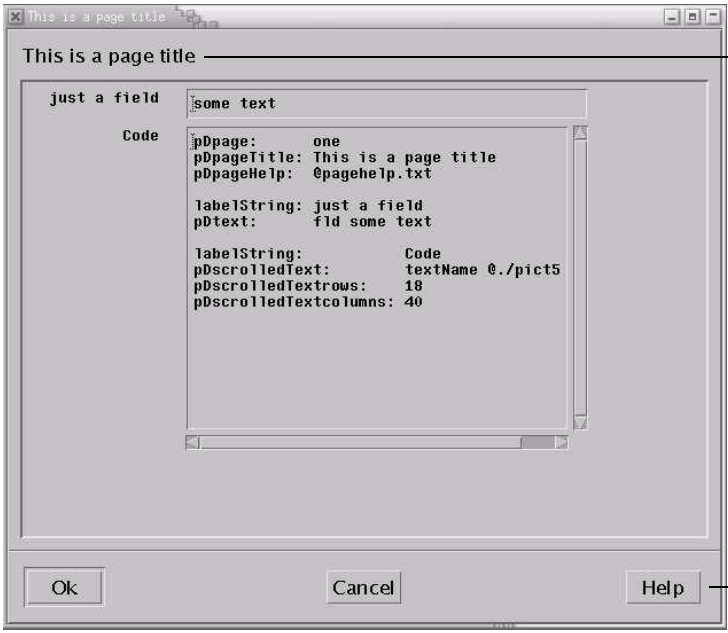
Images

You can add an image (XPM file) to each page using the `pDpagePixmap:` line. The image appears on the left of the dialog. See the figure “Sample single page prompt dialog with pixmap” (page 653).

Button labels

By default, button labels are “Ok”, “Cancel”, “Next>>”, and “<<Previous”. You can change the display for these labels using the `pDpageOkLabel:`, `pDpageCancelLabel:`, `pdPageNextLabel:`, and `pDpagePrevLabel:` lines respectively.

Figure 24
Sample prompt dialog with page title and page help



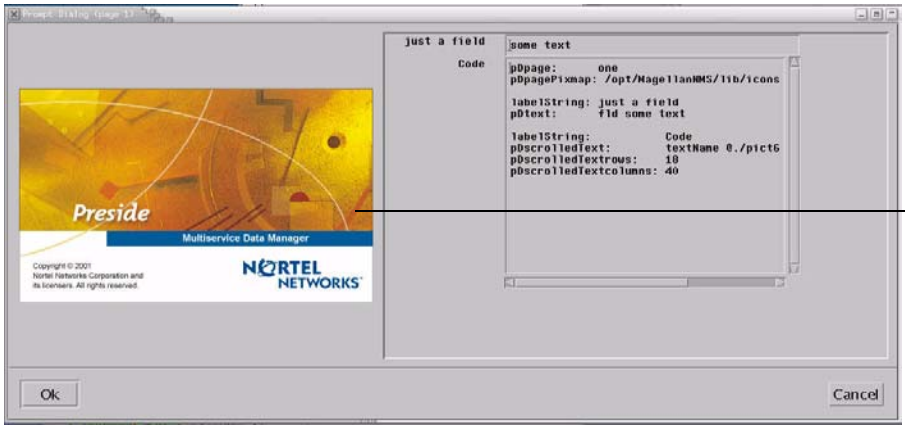
Description:

```
pDpage:          one
pDpageTitle:     This is a page title
pDpageHelp:      @pagehelp.txt

labelString:     just a field
pDtext:          fld some text

labelString:                Code
pDscrolledText:             textName @./pict5.txt
pDscrolledTextrows:         18
pDscrolledTextcolumns:      40
```

Figure 25
Sample single page prompt dialog with pixmap



Page pixmap

Description:

```
   pDpage:        one
   pDpagePixmap: \
      /opt/MagellanNMS/lib/icons/C/PresideSplash.xpm

   labelString:  just a field
   pDtext:       fld some text

   labelString:             Code
   pDscrolledText:          textName @./pict6.txt
   pDscrolledTextrows:      18
   pDscrolledTextcolumns:   40
```

Adding a Help button

You can associate help text or a help file with a page using the `pDpageHelp:` line. When a page with a specified `pDpageHelp:` line is active, the Help button appears on the corresponding dialog page. Clicking the Help button displays any associated text in a separate help dialog.

Table 6
promptDlog description file directives: page

Option	Presence	Description
pDpage: <page names...>	Optional	Identifies the page. There can be multiple name aliases, separated by spaces. If there is more than one page, the dialog appears in Wizard mode (with Next and Previous buttons). otherwise, only the Ok and Cancel buttons are present. The page names must be unique across the whole dialog.
pDpageRule: <KSH expr. to set pDnextPage>	Optional Multiple	Specifies one or more rules controlling which page will be visited next. Ksh code sets the pDnextPage variable to the name of the next page (see pDpageLast:). When all the fields in the page are validated, the Next button is enabled, When you click the Next button, the rules are performed in the specified order. The first rule that returns a non empty value for pDnextPage is activated. If there are no rules for the page and there are multiple pages in the dialog description, the next page specified in the description file is taken as the next page.
pDpageLast: <page name>	Optional	Specifies that this page is a possible terminating page. When all the fields in the page are validated, the Ok button is enabled and the Next button is disabled. The specified name is one of the page names declared in the pDpage: line for this page. This name corresponds to the name used to get to this page by the page rule used in the previous page. This allows a page to be terminal in some flows but intermediate in others.
pDpageStop: <page name>	Optional	Similar to pDpageLast: but does not forcibly disable the Next button. Use this option to make following pages optional.

(Sheet 1 of 3)

Table 6 (continued)
promptDlog description file directives: page

Option	Presence	Description
pDpageEnterAction: <KSH code>	Optional	Specifies a section Ksh code to execute when this page is being entered. In the code, you can use the value of `pDpageTransition:` This value is either `NEXT` or `PREV` if the invocation is the result of a Next or Previous (backtrack) action.
pDpageLeaveAction: <KSH code>	Optional	Specifies a section Ksh code to be executed when this page exits (including when the dialog is confirmed while this page is active). In the code, you can use the value of `pDpageTransition`. This value is either `NEXT`, `PREV`, or `OK` if the invocation is the result of a Next, Previous (backtrack), or Ok dialog button action.
pDpageTitle: <title string>	Optional	Specifies a title to display at the top of the dialog and in the window frame when the page is active. By default, there is no title and the window frame displays “Prompt Dialog” or “Prompt Dialog (Page <page number>)”
pDpagePixmap: <XPM pixmap file path>	Optional	Specifies an XPM bitmap to display in the dialog. The bitmap displays on the left with no title, on the top left with a title.
pDpageOkLabel: <label string>	Optional	Overrides the Ok button label for the page (multipage mode). The default is “Ok”.
pDpageCancelLabel: <label string>	Optional	Overrides the Cancel button label for the page (multipage mode). The default is “Cancel”.
pDpageNextLabel: <label string>	Optional	Overrides the Next button label for the page (multipage mode). The default is “Next>>”.
pDpagePrevLabel: <label string>	Optional	Overrides the Previous button label for the page (multipage mode). The default is “<<Previous”.

(Sheet 2 of 3)

Table 6 (continued)
promptDlog description file directives: page

Option	Presence	Description
pDpage<resource>: <value>	Optional Multiple	Specifies additional XmForm resources (the form in the dialog that contains the various fields for the page). <resource> is case sensitive.
pDpageHelp: <help text>|@<help file>	Optional	If specified, the dialog also has a Help button which, when clicked, displays the specified help information.
pDinclude: <Ksh code file>	Optional Multiple	Sources the specified Ksh code file to support action specifications. The file is sourced once when the description file is parsed.
pDcode: <Ksh code and function definitions> **pDcodeEnd:**	Optional Multiple	At any point in the description file, a pDcode: block can be used to add Ksh definitions. The included code is sourced into the running **promptDlog**. If the end marker, pDcodeEnd: is not included, the remaining description file is sourced in as code.
pDconfirmAction: <Ksh code>	Optional	Ksh code to be invoked when the dialog is confirmed by clicking the Ok button).
pDcancelAction: <Ksh code>	Optional	Ksh code to be invoked when the dialog is cancelled by clicking the Cancel button.

(Sheet 3 of 3)

Field descriptors

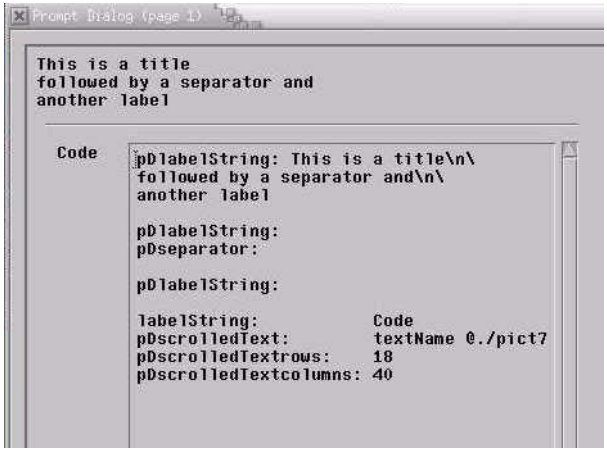
This section describes the following field descriptors:

The table "promptDlog description file directives: page" (page 654) provides a complete list of description resources applicable to a `pDpage:` block.

Title and separator

Titles and separators can be used to make the form more legible and provide in-line comments and prompts to the users. A title may be empty which causes an empty space to be displayed.

Figure 26
Sample title and separator



Description:

pDlabelString: This is a title\n\
followed by a separator and\n\
another label

pDlabelString:
pDseparator:

pDlabelString:

labelString: Code
pDscrolledText: *textName* @./pict7.txt
pDscrolledTextrows: 18
pDscrolledTextcolumns: 40

Table 7
promptDlog description file directives: title

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Specifies a label. An empty label creates a blank line in the form. See “Sample title and separator” (page 658).
pDlabelStringfontList: <font list specification>	Optional	Changes the title font.
pDlabel-String<resource>: <value>	Optional Multiple	Specifies additional `XmLabel` widget resources. <resource> is case sensitive.

Table 8
promptDlog description file directives: Separator

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Specifies a label with a separator that extends to the right of the label. An empty label creates a separator that extends the full width of the form.
pDseparator:	Mandatory	
pDseparator<resource>: <value>	Optional Multiple	Specifies additional `XmSeparator` widget resources. <resource> is case sensitive.

Text entry field

A text entry field prompts the user for a single line of text. The initial value for the text field is taken from either the input AVL's value for its variable name or from the description file.

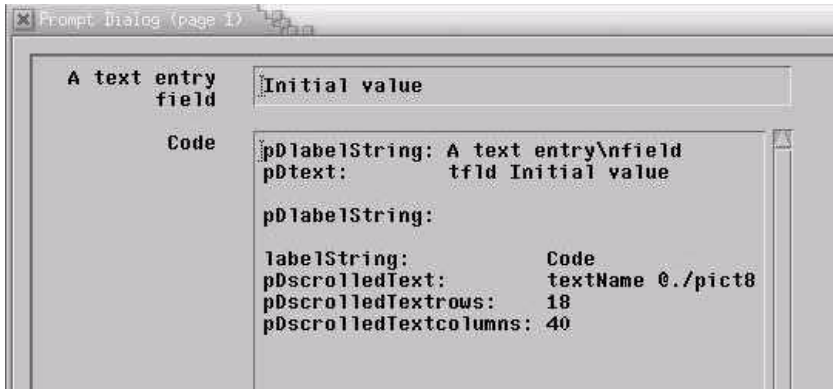
You can specify a set of Ksh patterns to validate the entered value. If the value is invalid when it loses keyboard focus, the following changes occur in the dialog to indicate the need for correction:

- The Ok button (single page mode) and Next button (multiple page mode) are disabled.
- The label background changes to red.
- An "X" mark appears to the left of the field name.
- A red outline borders the text field.

If you specify a help string or file, the right hand side of the field will have a Help button. Clicking the Help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure "Sample text entry field" (page 661) provides an example of the text entry field. The table "promptDlog description file directives: text entry field" (page 661) provides a list of descriptors for the text entry field.

Figure 27
Sample text entry field



Description:

pDlabelString: A text entry\nfield
pDtext: *tfld* Initial value

pDlabelString:

labelString: Code
pDscrolledText: *textName* @./pict8.txt
pDscrolledTextrows: 18
pDscrolledTextcolumns: 40

Table 9
promptDlog description file directives: text entry field

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDtext: <var. name> [<value>]	Mandatory	<var. name> is the AVL variable name for this field. Initial content is taken from the input AVL, if any, or the specified value.

(Sheet 1 of 2)

Table 9 (continued)
promptDlog description file directives: text entry field

Option	Presence	Description
pDtextPatterns: <KSH patterns>	Optional Multiple	Optional validation patterns (Ksh syntax). Multiple patterns can be specified (they are ORed together). If the contents do not match, the Ok button is disabled. Also, when the field loses keyboard focus, the field is marked invalid with an X suffix on the label and a red background/border.
pDtextcolumns: <nb characters>	Optional	Width in characters for the field.
pDtexteditable: **True**|**False**	Optional	Controls whether or not the field can be modified.
pDtext<resource>**:** <value>	Optional Multiple	Specifies additional XmTextField widget resources. <resource> is case sensitive.
pDtextHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button displays to the right of the field. Clicking the Help button displays the specified text

(Sheet 2 of 2)

Scrolled text field

This field is a read-only multi-line scrolled text field. You can control the number of columns and lines in field. This type of text field is useful for displaying extended information such as copyright notices and instructions.

The value for the text field is taken from either the input AVL's value for its variable name or from the description file. Unlike other fields, the value of a scrolled text field is not provided in the output AVL information. If you want to store the value of the text field, you need create a `pDConfirmAction:` action to extract the value from the `pDitemValues` associative array. For more information, see "Inline execution and action support calls" (page 700).

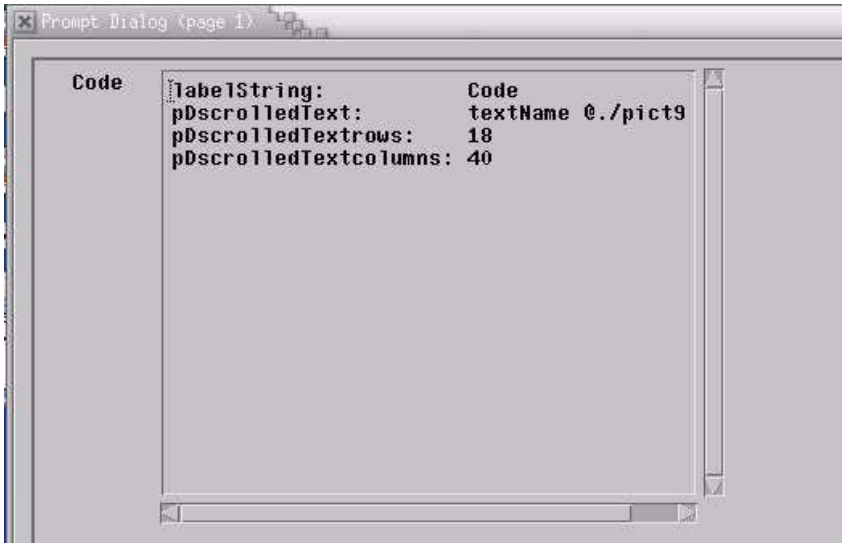
You can specify a set of Ksh patterns to validate the entered value. If the value is invalid when it loses keyboard focus, the following changes occur in the dialog to indicate the need for correction:

- The Ok button (single page mode) and Next button (multiple page mode) are disabled.
- The label background changes to red.
- An "X" mark appears to the left of the field name.
- A red outline borders the text field.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the Help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure "Sample scrolled text field" (page 664) provides an example of the scrolled entry field. The table "promptDlog description file directives: scrolled text" (page 665) provides a list of descriptors for the scrolled text entry field.

Figure 28
Sample scrolled text field



Description:

labelString:	Code
pDscrolledText:	*textName* @./pict9.txt
pDscrolledTextrows:	18
pDscrolledTextcolumns:	40

Table 10
promptDlog description file directives: scrolled text

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDscrolledText: <var. name> [@<file path>|<value>]	Mandatory	<var. name> is the AVL variable name for this field. Content is taken from the input AVL, if any, or the specified value.
pDscrolledTextcolumns: <nb characters wide>	Optional	Width in characters for the field.
pDscrolledTextrows: <nb characters high>	Optional	Height in characters for the field.
pDscrolled-Text<resource>**:** <value>	Optional Multiple	Additional XmText widget resources can be specified. <resource> is case sensitive.
pDscrolledTextHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button displays to the right of the field. Clicking the Help button displays the specified text.

Password field

This field is a specialized text field for entering a password. You can enter text in this field by typing, pasting, and dropping actions. Text entered in this field displays as asterisks (*). The password is kept internally and provided as the field's AVL value.

The initial value for the password field is taken from either the input AVL's value for its variable name or from the description file.

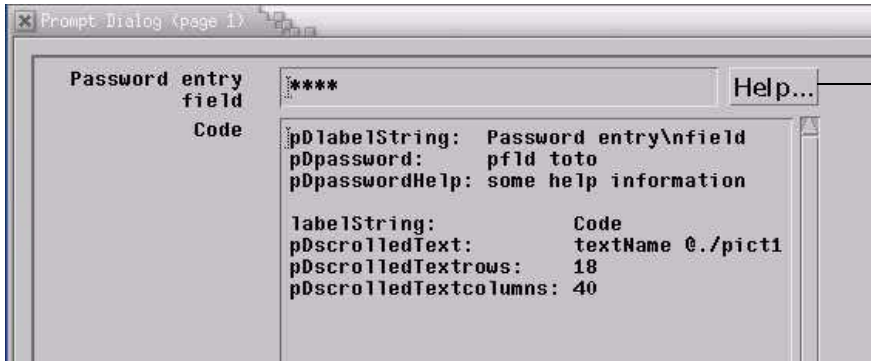
You can specify a set of Ksh patterns to validate the entered value. If the value is invalid when it loses keyboard focus, the following changes occur in the dialog to indicate the need for correction:

- The Ok button (single page mode) and Next button (multiple page mode) are disabled.
- The label background changes to red.
- An "X" mark appears to the left of the field name.
- A red outline borders the text field.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the Help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure "Sample password entry field with field Help button" (page 667) provides an example of the password field. The table "promptDlog description file directives: password field" (page 668) provides a list of descriptors for the password field.

Figure 29
Sample password entry field with field Help button



Description:

pDlabelString: Password entry\nfield
pDpassword: *pfld* toto
pDpasswordHelp: some help information

labelString: Code
pDscrolledText: *textName* @./pict10.txt
pDscrolledTextrows: 18
pDscrolledTextcolumns: 40

Table 11
promptDlog description file directives: password field

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDpassword: <var. name> [<value>]	Mandatory	<var. name> is the AVL variable name for this field. Initial content is taken from the input AVL, if any, or the specified value.
pDpasswordPatterns: <KSH patterns>	Optional Multiple	Optional validation patterns (Ksh syntax). Multiple patterns can be specified (they are ORed together). If the contents do not match, the Ok button is disabled. Also, when the field loses keyboard focus, the field is marked invalid with an X suffix to the label and a red background/border.
pDpasswordcolumns: <nb characters>	Optional	Width in characters for the field.
pDpasswordeditable: **True**|**False**	Optional	Controls whether or not the field can be modified.
pDpassword<resource>**:** <value>	Optional Multiple	Specifies additional XmTextField widget resources. <resource> is case sensitive.
pDpasswordHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

Component field

This field is a specialized text field for entering a component name. The entered value is provided as the following AVL values:

- `<var. name>` — the component ID in display format
- `<var. name>mod` — the component ID's module mnemonic name
- `<var. name>api` — the component ID in API format
- `<var. name><subcomp type>` — the value of each subcomponent level (non-links)
- `<var. name>type` — for non-link components, the concatenated (with '-') values of all the component ID's subcomponent types. For links, it is the link type
- `<var. name>ep1` and `<var. name>ep2` — for links, the component IDs of the both endpoints in API format

This field supports a popup menu. The popup menu is also available by clicking the drop-down arrow on the right side of the field. The popup menu contains the following commands:

- Get Context
  This command fetches the current MDM component context as the field value
- Select Matching Component...
  This command is enabled only if the field contents identifies a Passport component with an `EM` prefix. Selecting this command opens the Passport Component Selector (`ppCompSelector`) dialog which lists all matching components. For details on this dialog, see ppCompSelector in 241-6001-301 *Preside MDM Customization Administrator Guide*. You can specify Passport CAS CLI wild-cards, as well as module name level wild-cards, in this field. The dialog displays the matching components, if any, and lets you select the one to use for the field's value.

The initial value for the field is from one of two sources. The initial value in the field can be from the input AVL's value (the plain variable name with no suffix and whose value can be in API or display format) for its variable name. Or, the initial value can be from the description file.

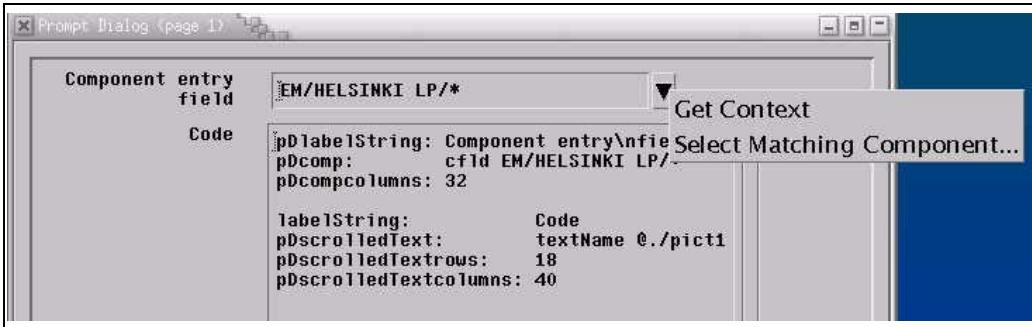
You can specify a set of Ksh patterns to validate the entered value in MDM component ID display format. If the value is invalid when it loses keyboard focus, the following changes occur in the dialog to indicate the need for correction:

- The Ok button (single page mode) and Next button (multiple page mode) are disabled.
- The label background changes to red.
- An “X” mark appears to the left of the field name.
- A red outline borders the text field.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure “Sample component entry field and field menu” (page 671) provides an example of the component entry field. The table “promptDlog description file directives: component field” (page 673) provides a list of descriptors for the component field.

Figure 30
Sample component entry field and field menu



Description:

```
pDlabelString:  Component entry\nfield
pDcomp:         cfld EM/HELSINKI LP/*
pDcompcolumns:  32

labelString:            Code
pDscrolledText:         textName @./pict11.txt
pDscrolledTextrows:     18
pDscrolledTextcolumns:  40
```

Figure 31
Sample component entry field (Passport component selector dialog)

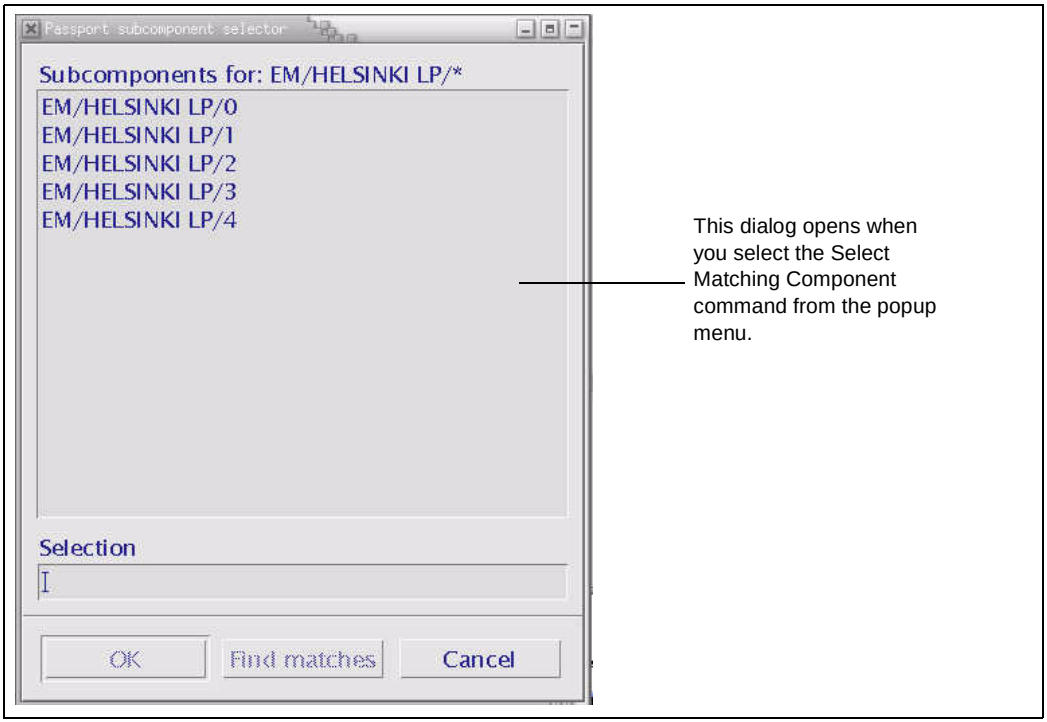


Table 12
promptDlog description file directives: component field

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDcomp: <var. name> [<value>]	Mandatory	<var. name> is the AVL variable name for this field. Initial content is taken from the input AVL, if any, or the specified value.
pDcompPatterns: <KSH patterns>	Optional Multiple	Optional validation patterns (Ksh syntax). The pattern is applied to the component ID as specified in the text field. Multiple patterns can be specified (they are ORed together). If the contents do not match, the Ok button is disabled. Also, when the field loses keyboard focus, the field is marked invalid with an X suffix to the label and a red background/border.
pDcompcolumns: <nb characters>	Optional	Width in characters for the field.
pDcompeditable: **True**|**False**	Optional	Controls whether or not the field can be modified.
pDcomp<resource>**:** <value>	Optional Multiple	Specifies additional XmTextField widget resources.<resource> is case sensitive.
pDcompHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

File field

This field is a specialized text field for entering a file name. The field also supports a Browse button. Clicking the Browse button opens a standard File Selection dialog in context to the current field value. Selecting a value from the dialog populates the field's value.

The initial value for the field is from either the input AVL's value for its variable name or from the description file.

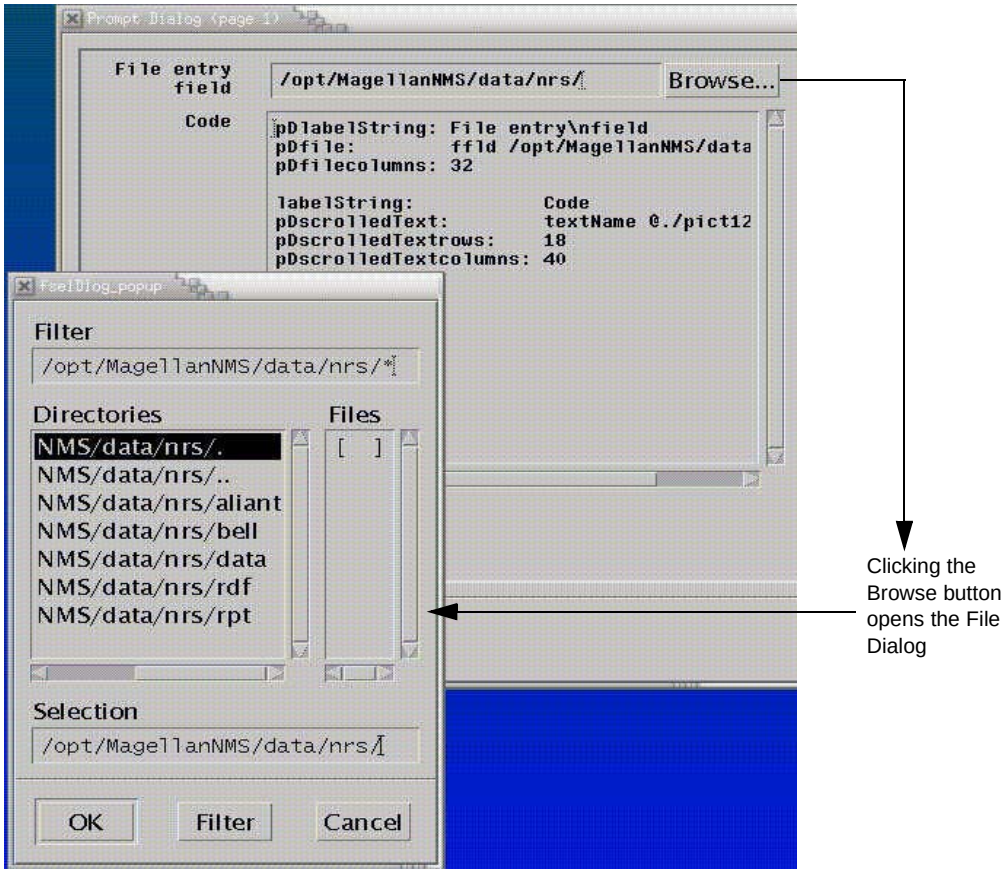
You can specify a set of Ksh patterns to validate the entered value. If the value is invalid when it loses keyboard focus, the following changes occur in the dialog to indicate the need for correction:

- The Ok button (single page mode) and Next button (multiple page mode) are disabled.
- The label background changes to red.
- An "X" mark appears to the left of the field name.
- A red outline borders the text field.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure "Sample file entry field" (page 675) provides an example of the file field. The table "promptDlog description file directives: file field" (page 676) provides a list of descriptors for the file field.

Figure 32
Sample file entry field



Description:

pDlabelString: File entry\nfield
pDfile: *ffld* \
/opt/MagellanNMS/data/nrs/data
pDfilecolumns: 32

labelString: Code
pDscrolledText: *textName* @./pict12.txt
pDscrolledTextrows: 18
pDscrolledTextcolumns: 40

Table 13
promptDlog description file directives: file field

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDfile: <var. name> [<initial file/directory>]	Mandatory	<var. name> is the AVL variable name for this field. Initial content is taken from the input AVL, if any, or the specified value. The value is also taken as the target directory/file for the File Browser dialog.
pDfilePatterns: <KSH patterns>	Optional Multiple	Optional validation patterns (Ksh syntax). Multiple patterns can be specified (they are ORed together). If the contents do not match, the Ok button is disabled. Also, when the field loses keyboard focus, the field is marked invalid with an X suffix to the label and a red background/border.
pDfilecolumns: <nb characters>	Optional	Width in characters for the field.
pDfileeditable: **True**|**False**	Optional	Controls whether or not the field can be modified.
pDfile<resource>**:** <value>	Optional Multiple	Specifies additional XmTextField widget resources. <resource> is case sensitive.
pDfileHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

Numerical range field

This field is a specialized text field for specifying a numerical value. The right side of the field contains a spin box with up and down arrows. The arrows let you increase or decrease the numerical value within a specified minimum and maximum range. You can also enter a value in the box by typing, pasting and dropping actions.

The initial value for the field is from either the input AVL's value for its variable name or from the description file. The minimum, maximum, and increment values can also be taken in from the input AVL (as the `<var. name>min`, `<var. name>max`, and `<var. name>inc` variables respectively) but are not provided in the output AVL.

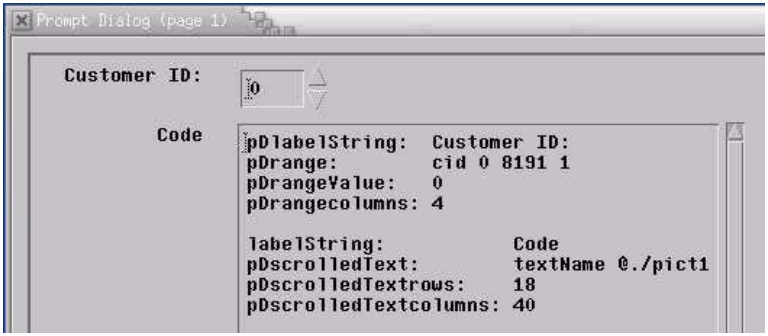
The field value is validated against its specified maximum, minimum, and increment. If the value is invalid when it loses keyboard focus, the following changes occur in the dialog to indicate the need for correction:

- The Ok button (single page mode) and Next button (multiple page mode) are disabled.
- The label background changes to red.
- An "X" mark appears to the left of the field name.
- A red outline borders the text field.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure "Sample numerical range entry field" (page 678) provides an example of the numerical range field. The table "promptDlog description file directives: numerical range" (page 679) provides a list of descriptors for the numerical range field.

Figure 33
Sample numerical range entry field



Description:

pDlabelString: Customer ID:
pDrange: *cid* 0 8191 1
pDrangeValue: 0
pDrangecolumns: 4

labelString: Code
pDscrolledText: *textName* @./pict13.txt
pDscrolledTextrows: 18
pDscrolledTextcolumns: 40

Table 14
promptDlog description file directives: numerical range

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDrange: <var. name> <minimum> <maximum> [<increment>]	Mandatory	<var. name> is the AVL variable name for this field. The minimum and maximum values are specified with the increment (1 by default). The contents can be specified manually but the field is considered valid only if the entered value is a number within the specified range and increment. If the field is considered invalid, the Ok button is disabled and marked with an X suffix to the label and a red background/border when it loses keyboard focus.
pDrangeValue: <value>	Optional	Initial value for the field if not specified in the input AVL.
pDrangecolumns: <nb characters>	Optional	Width in characters for the field.
pDrange<resource>: <value>	Optional Multiple	Specifies additional XmSpinBox widget resources. <resource> is case sensitive.
pDrangeHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

Date field

This field is a specialized field for specifying a date as three numerical range entry subfields for year, month, and day.

The right side of each subfield contains a spin box with up and down arrows. The arrows let you incrementally increase or decrease the value by one. You can also enter a value in the box by typing, pasting and dropping actions. The maximum and minimum values for the subfields are as follows:

- 1960–2100 for the year
- 1–12 for the month
- 1–31 for the day

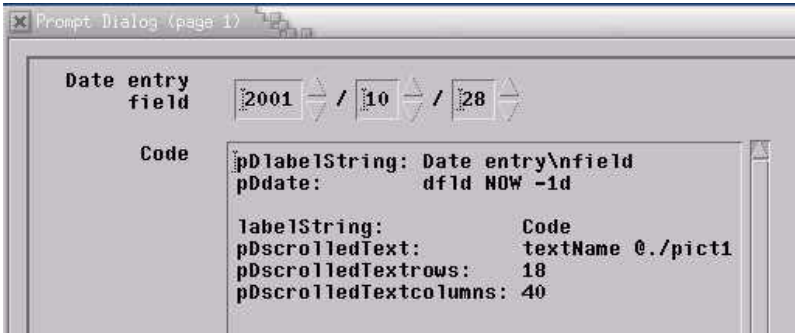
The initial value for the field is from either the input AVL's value for its variable name or from the description file. The output value for the field is a date in API format (`YYYY MM DD`). You can specify the input value in either the input AVL or the description file as the special value `NOW` or a date in API format. Optionally, you can provide an offset to the input value by following the value with a positive or negative offset specification in days, weeks, or years.

The field value is validated against the subfields' maximum and minimum. No validation of the correctness of the day against the month/year specification occurs. If the value is invalid when it loses keyboard focus, the following changes occur in the dialog to indicate the need for correction:

- The Ok button (single page mode) and Next button (multiple page mode) are disabled.
- The label background changes to red.
- An "X" mark appears to the left of the field name.
- A red outline borders the text field.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

Figure 34
Sample date entry field



Description:

pDlabelString: Date entry\nfield
pDdate: *dfld* **NOW -1d**

labelString: Code
pDscrolledText: *textName* @./pict14.txt
pDscrolledTextrows: 18
pDscrolledTextcolumns: 40

Table 15
promptDlog description file directives: data field

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDdate: <var. name> [**NOW**|<yyyy> <mm> <dd>] **[+|-**<offset>**D|W|Y]**	Mandatory	<var. name> is the AVL variable name for this field. Initial content is taken from the input AVL, if any, or the specified value. The value is stored in the AVL in API format (YYYY MM DD). If NOW is specified as the initial value, the current date is used. You can specify a positive or negative offset from the specified date in days (D), weeks (W), or years (Y). Otherwise, you can specify an API time format date as the initial value.
pDdate<resource>: <value>	Optional Multiple	Specifies additional XmRowColumn widget resources. <resource> is case sensitive.
pDdateHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

Date-time field

This field is a specialized text field for specifying a date and time as six numerical range entry subfields for year, month, day, hour, minutes, and seconds.

The right side of each subfield contains a spin box with up and down arrows. The arrows let you incrementally increase or decrease the value by one. You can also enter a value in the box by typing, pasting and dropping actions. The maximum and minimum values for the subfields are as follows:

- 1960–2100 for the year
- 1–12 for the month
- 1–31 for the day
- 0–23 for the hour
- 0–59 for the minutes
- 0–59 for the seconds subfields respectively.

The initial value for the field is either taken from the input AVL's value for its variable name or as specified in the description file. The input and output value for the field is a date-time in API format (`YYYY MM DD HH MM SS`). The input value (in the input AVL or the description file) can be specified as the special value `NOW` or a date-time in API format. That value can optionally be followed by a positive or negative offset specification in days, weeks, years, hours, minutes, or seconds.

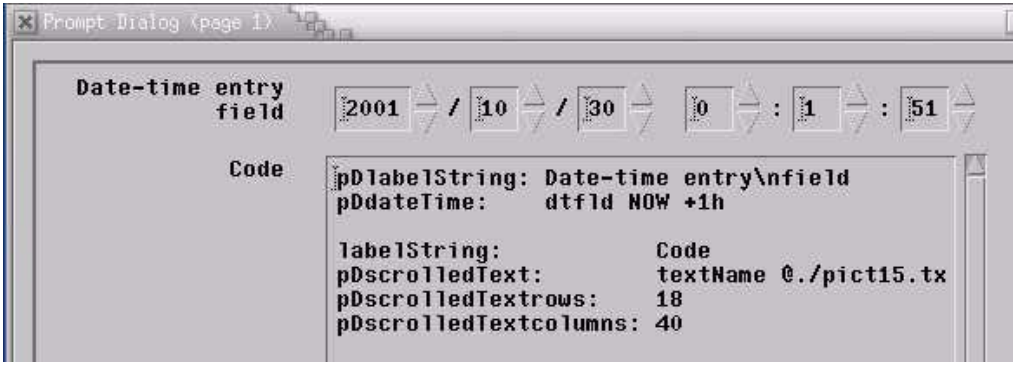
The field value is validated against the subfields' maximum and minimum. No validation of the correctness of the day against the month/year specification occurs. If the value is invalid when it loses keyboard focus, the following changes occur in the dialog to indicate the need for correction:

- The Ok button (single page mode) and Next button (multiple page mode) are disabled.
- The label background changes to red.
- An "X" mark appears to the left of the field name.
- A red outline borders the text field.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure “Sample date-time entry field” (page 684) provides an example of the date-time field. The table “promptDlog description file directives: date-time field” (page 685) provides a list of descriptors for the date-time field.

Figure 35
Sample date-time entry field



Description:

pDlabelString: Date-time entry\nfield
pDdateTime: *dtfld* **NOW +1h**

labelString: Code
pDscrolledText: *textName* @./pict15.txt
pDscrolledTextrows: 18
pDscrolledTextcolumns: 40

Table 16
promptDlog description file directives: date-time field

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDdateTime: <var. name> [**NOW**|<yyyy> <mm> <dd>] [**+**|**-**<off-set>**D**|**W**|**Y**|**H**|**M**|**S]**	Mandatory	<var. name> is the AVL variable name for this field. Initial content is taken from the input AVL, if any, or the specified value. The value is taken/stored in the AVL in API format (YYYY MM DD HH MM SS). If NOW is specified as the initial value, the current date and time is used. You can specify a a positive or negative offset from the specified date and time in days (D), weeks (W), years (Y), hours (H), minutes (M), or seconds (S). Otherwise, you can specify an API time format date and time as the initial value.
pDdateTime<resource>: <value>	Optional Multiple	Specifies additional XmRowColumn widget resources. <resource> is case sensitive.
pDdateTimeHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

Radio button box

This field provides a set of radio buttons for specifying a single atomic value from a set of individual values.

You describe the field by listing each radio button the field is to contain. Each button defines an AVL variable name (usually the same for each radio button of the same field), a toggle value (different for each radio button and used as

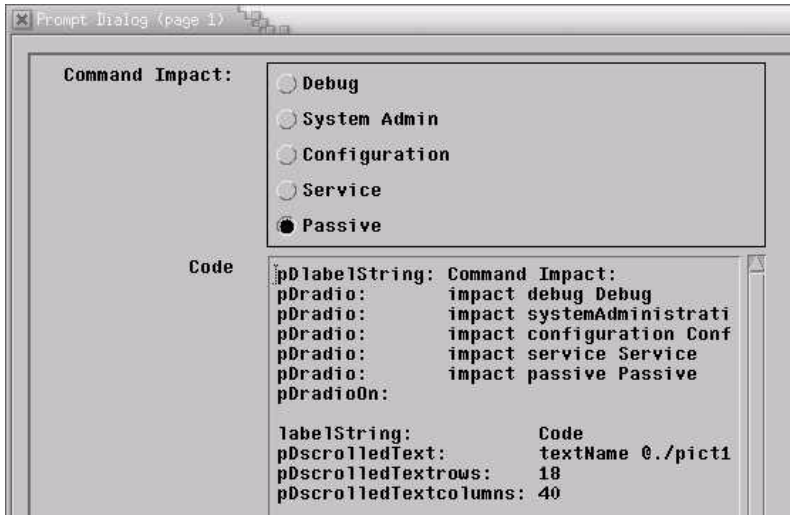
the value of the AVL variable when the button is selected), and a label string which is displayed. You can specify that there must always be at least one set button in the field (`pDradioalwaysOne:`).

The initial button set is either the one whose toggle name matches the variable name in the input AVL or the one followed by the `pDradioOn:` line in the description file.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure “Sample radio button box” (page 687) provides an example of a field containing radio buttons. The table “promptDlog description file directives: radio button box” (page 688) provides a list of descriptors for a radio button box field.

Figure 36
Sample radio button box



Description:

```
pDlabelString:  Command Impact:
pDradio:        impact debug Debug
pDradio:        impact systemAdministration \
                   System Admin
pDradio:        impact configuration \
                   Configuration
pDradio:        impact service Service
pDradio:        impact passive Passive
pDradioOn:

labelString:            Code
pDscrolledText:         textName @./pict16.txt
pDscrolledTextrows:     18
pDscrolledTextcolumns:  40
```

Table 17
promptDlog description file directives: radio button box

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDradio: <var. name> <toggle name> <label string>	Mandatory Multiple	Adds a radio button to the Radio Box. <var. name> is the AVL variable name for this field. It is typically the same for all radio buttons in the box. When the button is set, the AVL value is set to the corresponding <toggle name> value. <label string> is the label of the radio button as it appears in the display.
pDradioOn:	Optional	When specified, causes the preceding radio button to be set. If a <var. name> mapping exists in the input AVL, this is ignored in favour of the radio button whose <toggle name> matches the AVL value.
pDradioalwaysOne: true|false	Optional	If specified as true, forces the Radio Box to always have one of the buttons set.
pDradio<resource>: <value>	Optional Multiple	Specifies additional XmBulletinBoard (radio box) widget resources. <resource> is case sensitive.
pDradioHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

Check button box

This field provides a set of check buttons for specifying zero, one, or more atomic values from a set of individual values.

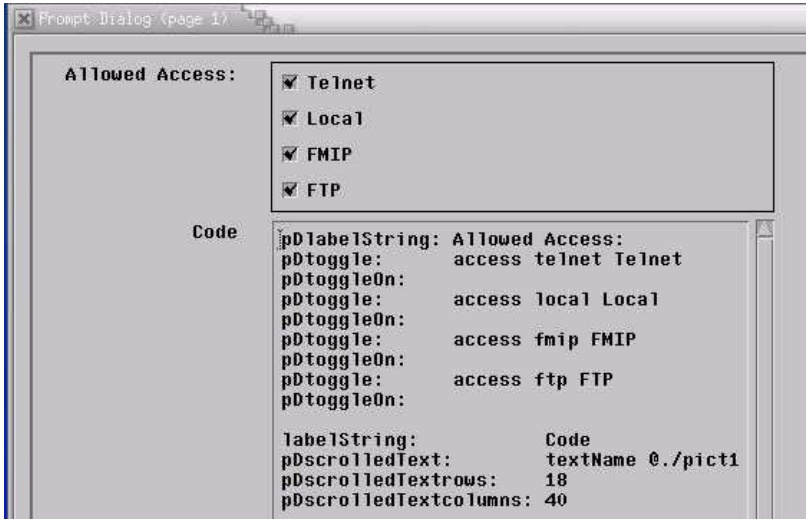
You describe the field by listing each check button the field is to contain. Each button defines an AVL variable name (usually the same for each check button of the same field), a toggle value (different for each check button), and a label string which is displayed. The AVL value for the field is the concatenation of the toggle names (separated by tabs) of each selected check button(s).

The initial buttons set are either the ones whose toggle names are listed in the matching variable name value in the input AVL or the ones followed by the `pDtoggleOn`: line in the description file.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure “Sample check button box” (page 690) provides an example of a field containing check buttons. The table “promptDlog description file directives: check button box” (page 691) provides a list of descriptors for the check button field.

Figure 37
Sample check button box



Description:

```
pDlabelString:  Allowed Access:
pDtoggle:       access telnet Telnet
pDtoggleOn:
pDtoggle:       access local Local
pDtoggleOn:
pDtoggle:       access fmip FMIP
pDtoggleOn:
pDtoggle:       access ftp FTP
pDtoggleOn:

labelString:            Code
pDscrolledText:         textName @./pict17.txt
pDscrolledTextrows:     18
pDscrolledTextcolumns:  40
```

Table 18
promptDlog description file directives: check button box

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDtoggle: <var. name> <toggle name> <label string>	Mandatory Multiple	Adds a check button to the check button box. <var. name> is the AVL variable name for this field. It is typically the same for all check buttons in the box. The AVL value is set to the <toggle name> values of all the set buttons in the box separated by a single blank. <label string> is the label of the check button as it appears in the display.
pDtoggleOn:	Optional	When specified, causes the preceding check button to be set. If a <var. name> mapping exists in the input AVL, this is ignored in favour of the check button whose <toggle name> values are listed in the AVL value.
pDtoggle<resource>: <value>	Optional Multiple	Specifies additional XmRowColumn widget resources. <resource> is case sensitive.
pDtoggleHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

List

This field provides a list of named entries from which one or more values can be selected.

You can specify individual list entries (`pDlistEntry:`) or use a file (`pDlistEntries:`) to populate the list. When using a file as input, the next line in the file that is not a comment line becomes an item in the list.

You can also populate the list with the names of Passport nodes or groups (`pDlistPassports:`). You can specify whether the list should be populated with the names of all known Passport groups or modules, or with the names of the modules within a specified group. As well, you can also specify the "`*`" wild-card value. These Passport items can be used as the values of the `@nodes` or `@start` **DoEPITemplate** special AVL values. When used, these special AVL values cause **DoEPITemplate** to act upon the specified nodes, `@nodes`, or to start a provisioning session on each one (`@start`).

The list can also be populated with the names of components from either the GMDR or Network Model servers. You can list modules or links matching a specific type or all types. And, you can list the next layer or all subcomponents of a specified component.

You can select from various types of lists: (`pDlistselectionPolicy:`):

- browsing list (`BROWSE_SELECT`) with a single selection with browsing behaviour
- single selection list (`SINGLE_SELECT`) with 0 or 1 selection
- multiple selection list (`MULTIPLE_SELECT`) with 0 or more selections
- extended selection (`EXTENDED_SELECT`) which allows 0 or more selections with selection extension behaviour

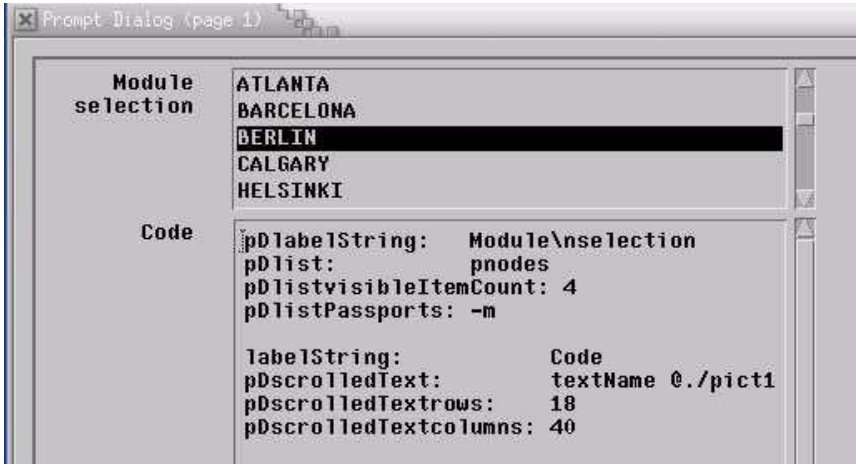
The AVL value of the field is the concatenation of the selected item strings separated by tabs. The initial selections match the values of the variable name in the input AVL.

Using `pDlistAction:,` you can specify a section of Ksh code to execute whenever a selection action occurs in the list. The current selections are available as the `pDlistSelectedItems` variable (tab separated). Lists often need some code support to achieve the necessary results. Refer to “Inline execution and action support calls” on page 700 for more information for the function calls that are provided to help.

If you specify a help string or file, the right side of the field will have a Help button. Clicking the help button displays the help text. If the label string is empty, the field extends to the left side of the dialog.

The figure “Sample list field” (page 694) provides an example of the list field. The table “promptDlog description file directives: list” (page 695) provides a list of descriptors for the list field.

Figure 38
Sample list field



Description:

pDlabelString: Module\nselection
pDlist: *pnodes*
pDlistvisibleItemCount: 4
pDlistPassports: -m

labelString: Code
pDscrolledText: *textName* @./pict18.txt
pDscrolledTextrows: 18
pDscrolledTextcolumns: 40

Table 19
promptDlog description file directives: list

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDlist: <var. name>	Mandatory	<var. name> is the AVL variable name for this field. Initial list selection is taken from the input AVL. The AVL value for the field is the value of the selected item (or the items separated by blanks for multiple select lists).
pDlistItem: <label string>	Optional Multiple	Adds the specified entry to the list. The string is both the displayed label and the AVL value of the field if this item is selected.
pDlistItems: <file path>	Optional Multiple	Adds the contents of the specified file to the list. The value of each non-commented (#) line is both the displayed label and the AVL value of the field if the corresponding item is selected.
pDlistPassports: -g [**-m**] |<group name> [*****]	Optional Multiple	Adds Passport nodes or groups to the list. If -g is specified, all known groups are added (with a @ prefix suitable to be used as the value of the special @nodes or @start AVL variable of **DoEPITemplate**). If -m is specified, then all the Passport node names are listed. If a group name is specified, all the Passports within that group are listed. If an * is specified, * is added to the list as a wild-card specification.

(Sheet 1 of 3)

Table 19 (continued)
promptDlog description file directives: list

Option	Presence	Description
pDlistComps[**-nm**] **-m**|**-l** [<type>] | **-next**|**-all** <component>		Adds component names from the Network Model (with -nm) or GMDR. If -m or -l is specified, only modules or links (of the specified type if any) are listed. If -next or -all is used, the next layer or all subcomponents of the specified component are listed. The component can be specified in API or display format.
pDlistselectionPolicy: <**BROWSE_SELECT**|**SINGLE_SELECT**|**MULTIPLE_SELECT**|**EXTENDED_SELECT**>	Optional	Specifies the list behaviour; BROWSE_SELECT (default) lets you select with browsing behaviour; SINGLE_SELECT supports one or no selection; MULTIPLE_SELECT supports multiple or no selection; EXTENDED_SELECT supports multiple or no selection with the ability of extending the selection of contiguous items.
pDlistvisibleItem-Count: <nb items>	Optional	Specifies the displayed height of the list as the number of items that should be visible. More items than specified may actually appear because of the space reserved for the horizontal scroll bar.
pDlistAction: <Ksh code>	Optional	Invokes the specified code when a list item is selected (only applicable for single and browse selection lists). In that code, the value of the pDselectedListItem: variable is available with the value of the of the selected item(s) (multiple items are tab separated).

(Sheet 2 of 3)

Table 19 (continued)
promptDlog description file directives: list

Option	Presence	Description
pDlist<resource>: <value>	Optional Multiple	Specifies additional XmList widget resources. <resource> is case sensitive.
pDlistHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

(Sheet 3 of 3)

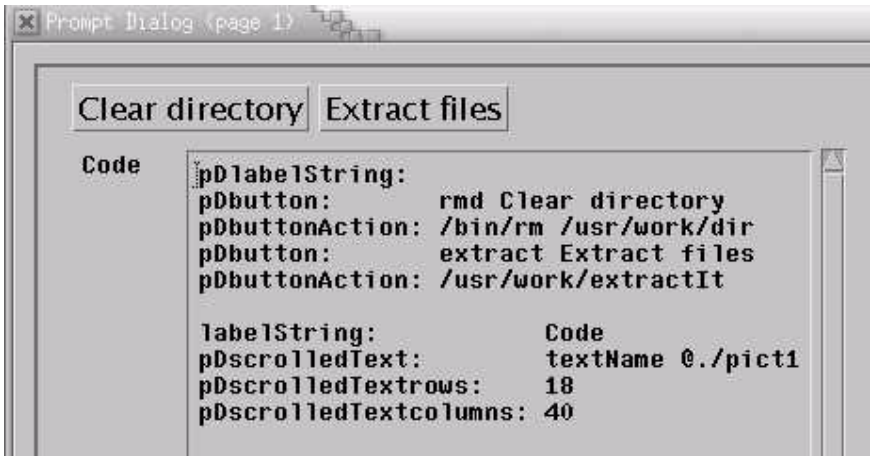
Push button box

This field provides one or more buttons that execute selected KSH code. If you use multiple buttons, they appear horizontally in a XmRowColumn widget. For every button, you need a `pDbuttonAction:` resource line to define the button's action.

Buttons are normally selectable. However, you can control the button though code using the `pDsetValue` function call. For more information, see "Inline execution and action support calls" (page 700).

The figure "Sample push button box" (page 699) provides an example of a field containing push buttons. The table "promptDlog description file directives: push button box" (page 700) provides a list of descriptors for the push button field.

Figure 39
Sample push button box



Description:

```
pDlabelString:
pDbutton:          rmd Clear directory
pDbuttonAction:    /bin/rm /usr/work/dir
pDbutton:          extract Extract files
pDbuttonAction:    /usr/work/extractIt

labelString:               Code
pDscrolledText:            textName @./pict19.txt
pDscrolledTextrows:        18
pDscrolledTextcolumns:     40
```

Table 20
promptDlog description file directives: push button box

Option	Presence	Description
pDlabelString: [<label string>]or **labelString:** [<label string>]	Mandatory	Label to the left of the field.
pDbutton: <var. name> <button name> <label string>	Mandatory Multiple	Adds a push button to the button box. <var. name> is the AVL variable name for this button. There is no output AVL value for this field.
pDbuttonAction: <KSH code>	Mandatory Multiple	Specifies the action invoked when the previously described button is pressed.
pDbutton<resource>: <value>	Optional Multiple	Specifies additional XmRowColumn widget resources. <resource> is case sensitive.
pDtoggleHelp: <help text>|@<help file> or **pDHelp:** <help text>|@<help file>	Optional	If specified, a Help button is added to the right of the field. Clicking the Help button displays the specified text.

Inline execution and action support calls

The promptDlog utility provides several function calls to help in the creation of the various action Ksh code and in the use of the utility in inline mode. These calls let you extract and set the current values of field controls (AVL) and to control the construction, display, and control of the dialog itself (inline mode).

Table 21
Inline mode and action support calls

Signature	Description
Field value extraction	
*${***pDitemValues***[*<variable name>*]}*	Not a function call in itself, but an associative array indexed by the control variable name whose entry value is the current value for the corresponding field (if any). Do not modify this field directly because the corresponding field will not be updated. Instead, use the **pDsetValue** and related function calls.
pDgetListItems <list var. name>	Populates the pDlistItemValues associative array with the values contained in the specified list.
Field value setting	
pDsetValue <var. name> <value>	This is a whole family of calls (described individually below) depending on the type of target field. If no field is associated with the named variable (hidden variable from the input AVL or dynamically added), its value is directly changed as specified. If a field is associated with the variable, the displayed value is also changed accordingly and any related validation checks are performed.
pDsetValue <text, password, component, or file field var. name> **--** | [**++**] <value>	**pDsetValue** call for text, password, component, and file fields. If -- is specified, the field is simply cleared. If a value is specified, it replaces the field value unless ++ is used in which case the value is appended to that of the field.
pDsetValue <scrolled text var. name> **--** | [**++**] <value> | @<file path>	**pDsetValue** call for scrolled text fields. If -- is specified, the field is simply cleared. If a value is specified, it replaces the field value unless ++ is used in which case the value is appended to that of the field. If a file path is specified with a @ prefix, the contents of the file always replaces the current contents of the field.
pDsetValue <range var. name> <value> [<min> <max> [<increment>]]	**pDsetValue** call for range fields. Simply sets the field to the specified value. If they are specified, the minimum, maximum and increment (defaults to 1) can also be changed.

(Sheet 1 of 4)

Table 21 (continued)
Inline mode and action support calls

Signature	Description
pDsetValue <date var. name> [**NOW**|<YYYY MM DD>] [**+**|**-** <offset>**D**|**W**|**Y**]	**pDsetValue** call for date fields. If NOW is specified, the current date is used else the specified API date is used. If an offset is specified, the current or specified date is first offset positively or negatively before being set.
pDsetValue <date var. name> [**NOW**|<YYYY MM DD HH MM SS>] [**+**|**-** <offset>**D**|**W**|**Y**|**H**|**M**|**S**]	**pDsetValue** call for date-time fields. If NOW is specified, the current date-time is used else the specified API date-time is used. If an offset is specified, the current or specified date-time is first offset positively or negatively before being set.
pDsetValue <radio or check var. name> [**!**] <toggle names...>	**pDsetValue** call for radio or check box fields. If ! is indicated, all the current values are first reset. Then, the named radio or check buttons are set.
pDsetValue <list var. name> -- [<items to delete...>] **| ++|+?** <items to add...> **| +@** <item file to add> **|** [**!**] <items to select...>	**pDsetValue** call for list fields (see also **pDsetListPassports** and **pDsetListComps**). If -- is specified, the following list items (pattern matched) are removed from the list. If no items is provided, the entire list is cleared. If ++ is used, the specified items are added at the end of the list. If +? is used, the items are added but only if they are not yet in the list. If +@ is used, the specified file's contents are added to the end of the list. Otherwise, the call will control the selection of the named items. If ! is specified, all items are first deselected. The following items are then selected.
pDsetValue <button var. name> **True**|**False**	**pDsetValue** call for button fields. This controls whether the button is selectable (**True**) or not (**False**).

(Sheet 2 of 4)

Table 21 (continued)
Inline mode and action support calls

Signature	Description
pDsetListPassports <list var. name> [**++**|**+?**] [**-m** [**-g**]]|<group> *****	Populates the named list with Passport group/module information such as the `pDlistPassports:` description file line. If ++ is used, the new items are added to the end of the list. If +? is used, they are added but only if they are not yet there. Otherwise, the new items will replace the current ones. If `-m` is specified, the names of all Passport modules known to HGDS are added. If `-g` is used, the names of all Passport groups are added. If a group is named, the modules within that group are added. If * is specified, the special wild-card (*) group name is added.
pDsetListComps <list var. name> [**++**|**+?**] [**-nm**] **-m**|**-l** [<type>] | **-all**|**-next** <component>	Populates the named list with Passport group/module information such as the `pDlistComps:` description file line. If ++ is used, the new items are added to the end of the list. If +? is used, they are added but only if they are not yet there. Otherwise, the new items will replace the existing ones. If `-nm` is specified, the information will come from the service selected Network Model, otherwise it will come from the GMDR server. If `-m` or `-l` is used, the modules or links matching the specified type (or all) will be added. If `-all` or `-next` is used, the next layer of subcomponents or all subcomponents of the specified component or subcomponent are added.
Inline mode commands	
pDinitX <command line args>	Initializes the X-Windows environment (XtInitialize). The command line arguments to the script are normally passed on to this function. The resulting top-level application shell widget is available as the `${pDTOPLEVEL}` and `${TOPLEVEL}` variable. Note: if you provide your own version of this function, ensure that you assign the top-level widget handle to a variable named `TOPLEVEL` for the DtKsh extensions to work properly.

(Sheet 3 of 4)

Table 21 (continued)
Inline mode and action support calls

Signature	Description
pDbuildDialog <top-level widget name> <description file> [<output AVL file>]	Initializes the dialog within the top-level application shell widget (as created by initX) and reads in the specified description file. If an output file is specified, the output AVL will be output to it instead of the standard output stream. The resulting dialog is available as the `${pDDLOG}` variable.
pDraiseDialog	Displays the dialog.
XtRealize <top-level widget name>	Realizes the application so it can interact with the user. Must be called before jumping into the main loop.
XtMainLoop	X-Windows event loop. Never return, all processing is the result of callbacks.
(Sheet 4 of 4)	

Example: Passport User Authentication

The sample prompt dialog descriptor file (`pwdDlog.desc`) in this section queries the parameters needed to concurrently create or modify a user authentication on multiple Passport nodes using the **DoEPITemplate** utility.

This dialog has two pages. The first page prompts for the target Passport group and the authentication information needed to connect and configure its nodes. See the figure "User authentication entry form, first page (Passport access)" (page 705). The second page requests the information on the user access privileges. See the figure "User authentication entry form, second page (user parameters)" (page 706). Also included is a typical implementation of the necessary EPI template (`auth.tmpl`) and how all these pieces work together.

The example uses the special `@nodes`, `@uid`, and `@pwd` AVL variables of the **DoEPITemplate** utility. The **DoEPITemplate** utility uses their values to connect to the specified group/node for configuration purpose. Another method to combine the prompt dialog with **DoEPITemplate** is the use the **DoEPITemplate `-dlog`** option with the path of the description file.

Figure 40
User authentication entry form, first page (Passport access)

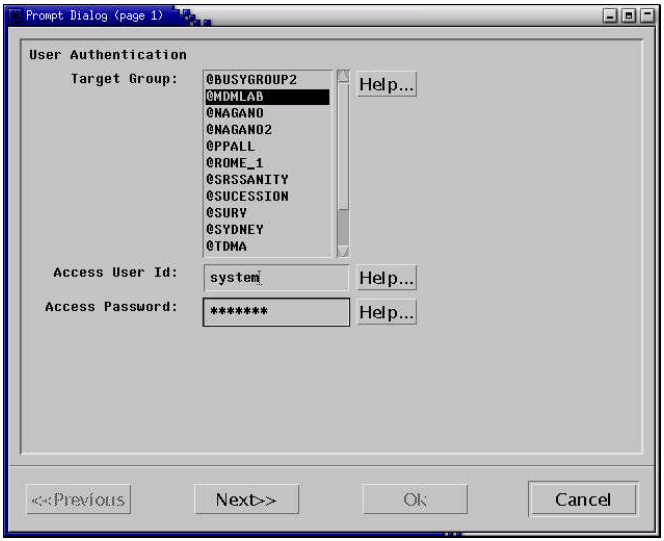
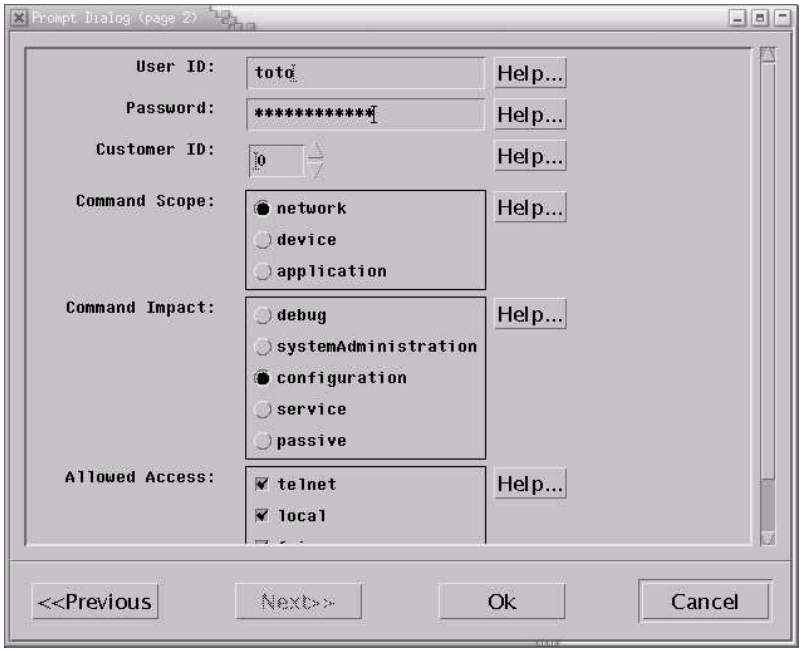


Figure 41
User authentication entry form, second page (user parameters)



The pwdDlog.desc prompt dialog description file:

```
# First page, information needed to connect
# to a group of Passports in provisioning mode
# and change their configuration
pDPage: auth

labelString: User Authentication

# Target Passport group.
# @start is a special DoEPITemplate AVL variable
# indicating which nodes to start a provisioning
# session on. pDlistPassports: -g * requests the
# list to be populated with all known group names and
# the wild card group. BROWSE_SELECT allows the
# selection of on item at a time.
labelString: Target Group:
pDlist: @start
```

```
pDlistPassports: -g *
pDlistselectionPolicy: BROWSE_SELECT
pDlistvisibleItemCount: 10
pDhelp: Passport groups on which nodes to \
create/change the user.

# User ID and Password used to connect to the
# Passports (the special DoEPITemplate AVL variables
# @uid and @pwd are used). The password pattern
# requires that at least 5 characters be specified.
labelString:    Access User Id:
pDtext:         @uid
pDtextcolumns:  8
pDtextPatterns: +(?)
pDhelp: User Name to access and reconfigure the \n\
Passports nodes (1 to 8 ASCII characters)

labelString:   Access Password:
pDpassword:    @pwd
pDtextcolumns: 16
pDpasswordPatterns: ?????*(?)
pDhelp: Password to access and reconfigure the \n\
Passports nodes (5 to 8 ASCII characters)

# The second page contains the specification of the
# authentication to add/change.
pdPage: data

labelString:    User ID:
pDtext:         uid
pDtextcolumns:  8
pDtextPatterns: +(?)
pDhelp: User Name (1 to 8 ASCII characters)

labelString:        Password:
pDpassword:         pwd
pDtextcolumns:      16
pDpasswordPatterns: ?????*(?)
pDhelp: Password (5 to 8 ASCII characters)

labelString:    Customer ID:
pDrange:        cid 0 8191
pDrangecolumns: 4
```

```
pDhelp: Customer Identification number ([0-8191])

labelString: Command Scope:
pDradio:     scope network     network
pDradio:     scope device      device
pDradio:     scope application application
pDradioOn:
pDhelp: Allowed command scope for the user.

labelString: Command Impact:
pDradio:     impact debug   debug
pDradio:     impact systemAdministration \
systemAdministration
pDradio:     impact configuration configuration
pDradio:     impact service service
pDradio:     impact passive passive
pDradioOn:
pDhelp: Allowed command impact for the user.

labelString: Allowed Access:
pDtoggle:    access telnet telnet
pDtoggleOn:
pDtoggle:    access local  local
pDtoggleOn:
pDtoggle:    access fmip   fmip
pDtoggleOn:
pDtoggle:    access ftp    ftp
pDtoggleOn:
pDhelp: Allowed management interfaces.
```

Typical implementation of an authentication template (auth.tmpl):

```
# Passport User authentication template suitable
# for adding/modifying user authentications from
# doEPITemplate.
# AVL parameters:
#  mod        -- module name (from DoEPITemplate)
#  user       -- user name (1-8 chars)
#  pwd        -- password (5-8 chars)
#  cid        -- customer ID (0-8192)
#  scope      -- command scope
#                        (network, device, appl)
#  impact     -- command impact
```

```
#                          (debug, sysAdmin, config,
#                           service, passive)
#  access     -- allowed access
#                          (local, telnet, fmip, ftp)
# @TRY the adding of the user so the template behaves
# like a modify if the user already exists
@TRY $mod add access user/$user
$mod set user/$user password     $pwd
$mod set user/$user cid          $cid
$mod set user/$user scope        $scope
$mod set user/$user impact       $impact
$mod set user/$user allowedAccess $access
```

Typical invocation of the dialog and DoEPITemplate:

```
# raise a prompt dialog for the necessary information
# which returns 0 if the user confirmed the dialog.
if /opt/MagellanNMS/bin/promptDlog \
        -desc /opt/MagellanNMS/cfg/pwdDlog.desc \
        -out /tmp/auth.avl
then
   # perform the template on the resulting AVL
   # redirecting its output and error streams to
   # a text window.
   /opt/MagellanNMS/bin/DoEPITemplate -private \
            -async -foreach -trace \
            -load CURRENT -save CURRENT \
            -activate -avl /tmp/auth.avl \
            -tmpl /opt/MagellanNMS/cfg/auth.tmpl \
            2>&1 \
   | /opt/MagellanNMS/bin/xmsg - -delay 0

   # remove the AVL file once done
   /bin/rm /tmp/auth.tmpl
fi
```

Alternatively, the prompting dialog can be invoked through DoEPITemplate itself with its -dlog option:

```
# raise a prompt dialog for the necessary information
# and perform the template with it.
/opt/MagellanNMS/bin/DoEPITemplate -private \
        -async -foreach -trace \
```

```
            -load CURRENT -save CURRENT -activate \
            -dlog /opt/MagellanNMS/cfg/pwdDlog.desc \
            -tmpl /opt/MagellanNMS/cfg/auth.tmpl \
            2>&1 \
    | /opt/MagellanNMS/bin/xmsg - -delay 0
```

Index

F

G

H

M

N

P

Q

R

S

T

U

W

X

Preside Multiservice Data Manager

Embedded Programming Interface

Reference Guide

Release: R14.3

Publication: 241-6001-211
Document status: Standard
Document version: 14.3RSUP
Document date: December 2003
Printed in Canada

NORTEL NETWORKS